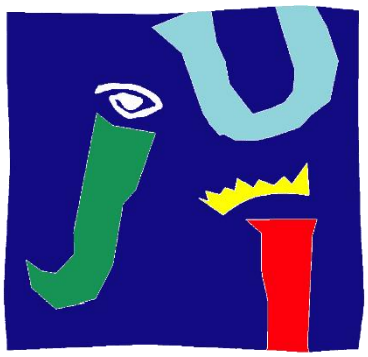

UNIVERSITAT JAUME·I

University Jaume I

Design and Development of Video Games Degree

Technical Report of the Final Degree Project

3D Puzzle Mobile Game Powered by Gyroscope And/Or Accelerometer

Student: **Antonio José Luque Ocaña**
Tutor: **Miguel Chover Selles**

“Angry Birds is a very simple idea but it’s one of those games that I immediately appreciated when I first started playing, before wishing that I had been the one to come up with the idea first.”

— Shigeru Miyamoto

Summary

This document presents the technical report for a Final Degree Project related to the Degree in Design and Development of Video Games. The project to develop consists on a three-dimensional videogame for android devices developed with Unity3D [1]. Its singular feature consists on taking advantage of accelerometer and gyroscope controllers integrated into android devices to rotate the platforms of the game levels. Mainly, the game will use gyroscope, and if the device has not this controller, the game will be able to employ accelerometer to emulate the same behavior.

Modern mobile devices have Inputs different to the rest of present devices able to run software. Inquiring into them, this project aims to develop a game for Android mobile devices that make special use of gyroscope and accelerometer sensors, in addition to the usual touch sensor. To carry it out, models and animations will be developed in 3D, 2D graphics, and especially C# programming, all original and functional through Unity3D game engine.

The game will be set in the north pole, with penguins, icebergs, realistic water, etc. In the game, the player can vary the rotation of ice platforms on which there will be penguins that slide and objects with which penguins can interact. The objective is to throw them all to the water, except for the protagonist penguin. In the numerous levels that are going to be developed, the shape of the platforms will change and new penguins and objects with different characteristics will appear. Levels will be short duration, as is demanded by players of today's mobile devices. For this last statement, take as an example the famous game Angry Birds [2], whose numerous levels are around 3 minutes in length.

The processing of data provided by the accelerometer and gyroscope will be such that the player will be able to select which of the two sensors to use and achieve an almost identical gaming experience. Facing also the challenge of developing on Android with Unity3D, it will be implemented a solution for the delay issue in the sound reproduction that this engine generates when compiles apps [3]. The facilities granted by the engine to save data will not be used, but code will be implemented to save data locally in encrypted form. Thanks to this, players will not be able to manually edit their save files and cheat their data. Additionally, the game will take full advantage of the particle system that the engine offers, as it is very powerful and visually attractive. Finally, it is intended to develop the feature of communicating with modern social networks, with the intention of being able to reach a viral marketing.

Key Words: App, Interaction, Measurement, Gyroscope, Accelerometer

Index

Figures - Index

1. Introduction

1.1 Project Objectives

The main objective of the project is to get a fully functional 3D puzzle game for Android mobile devices, by employing the knowledge acquired in a great diversity of subjects of the Degree in Design and Development of Video Games.

The developed game has the main objective of being able to reach the widest number of players possible. That is why it will need to be developed with the following characteristics:

- Being able to be played in the most common game device of nowadays: Mobile phones.
- Having intuitive controls for most players: In this case, it will be controlled by simply rotating the mobile phone, using the phone gyroscope.
- Considering players with low-end mobile devices:
    - Low-powered mobile phones: The game will work with a low number of Draw Calls [4], in search of performance.
    - Mobile phones without gyroscope: Accelerometer will be able to be used instead, emulating gyroscope's same behavior as much as possible. "Most Android-powered devices have an accelerometer, and [just] many now include a gyroscope" [5].
- Attempting to achieve a viral marketing: Players will be able to "share" the game download link in most commons social networks from inside the game in exchange of game rewards.
- Being nice-looking for most players: Let's be honest, cute animals are nowadays in fashion, so the game characters will be cute penguins modeled and animated for the project.
- Being lingually understandable for most players: The game texts will be shown in a language or other depending on the local device language settings, by reading a CSV localization file.

In order to accomplish all of this, I will write a Game Design Document, investigate how to perform some tasks still unfamiliar to me, program a lot of C# code, model and animate in 3D, draw in 2D, edit and adjust audio clips, merge all of them inside a game engine, test and correct all the work, and finally write this Technical Report.

1.2 Videogame Requirements

Stablishing a clear number of requirements of the final result is crucial in order to delimit the direction of the development process. These points below are technically required for making it possible to achieve the objectives described above.

- No bugs: The game should be fully played without encountering unexpected events and without crashing at any point.
- No cheatable: The game save data implementation should store the player game status in an encrypted file in order to keep protection against players who attempt to rewrite some parameters to cheat their game.
- Synchronized sound effects: Unity3D, the game engine employed in this project, generates a light but noticeable delay when playing audio clips when running a built android apk [3]. This issue must be solved for obtaining a nice quality game.
- Different device, same game: The same apk file must be able to be played with different kinds of android devices (with or without gyroscope, high or low powered, etc), experiencing a nice and similar gameplay.
- The game must be programmed efficiently and run fluid: Working code is not enough, it must also be clean and efficient.

1.3 General Environment of the Project

The game has been developed with the Operating System Windows 10 Home, under an Asus laptop purchased in 2013, with the following characteristics:

- Processor: Intel® Core™ i3-2350M / 2.3 GHz
- Ram Memory: 4 GB (3,89 GB available)
- OS of 64 bits
- Nvidia Gforce 610M 1GB

Tools:

- Unity3D:

  The game engine used for the entire project.

- Monodevelop:

  The integrated development environment employed, native of Unity3D. Code is written in C#.

- 3D Studio Max:

  The 3D computer graphics program used to create and animate models.

- Adobe Photoshop:

  The raster graphics editor used to create the title screen and the app logo.

- Audacity:

  The digital audio editor employed to edit and adjust every audio clip of the game.

- Camtasia Studio:

  The digital video editor and recorder used to create the promotional video of the game.

- Overleaf:

  The online LaTeX based word processor employed to write the Technical Proposal of this project.

- Microsoft Office Word:

  The word processor used to write this Final Report.

2. Planning

After defining the project and its requirements, we proceed to split all the work in tasks and to measure approximately how much time each task is going to take. This is the first approach, taken on February, and it will be compared with the real time employment once the project is finished, at the end of this Technical Report.

Tasks	Hours
1- Technical Proposal	
1.1- Attend classes about how to prepare the technical proposal	8
1.2- Write the technical proposal and elaborate it with LaTeX	6
2- Gyroscope	
2.1- Find information about how to integrate gyroscope usage with Unity3D on Android	2
2.2 - Develop a demo in which you can rotate a floating platform by rotating the device	10
2.3- Put limits on the rotation of the platform and reduce its speed, regardless of how much more the device rotates	1
2.4- Make the rotation to be relative at any angle. In other words, if you are playing standing up and you lie down, you can "recalibrate" and continue to play lying down	5
3- Physics	
3.1- Expand the demo, including spheres on the platform which move with the inclination of this	1
3.2- Replace the "gravity" of Unity3D as a reason of movement for a more precise auxiliary system, in which the objects on the platform always move on a horizontal plane but are displayed at one height or another on the platform, making use of Raycast	6
4- 3D Models	
4.1- Perform the 3D model of a penguin with 3D Studio Max	20
4.2- Put skeleton, create and export several animations for that penguin	7
4.3- Give iceberg texture to the demo platform	1
4.4- Find information about using realistic water in Unity3D and incorporate it, adapting the water fresnel to the scene	3
4.5- Add all models and animations to the demo with Unity3D	2
5- Physics (2nd part)	
5.1- Create the mechanics of "ice cubes", objects implemented so that the only force that can move them is the push of a penguin	3
5.2- Manually adjust the LookAt vector of the penguins at each moment, tending gently to look at where they are moving. Once a penguin falls into	4

the water, select where to look and implement it to swim away in that direction automatically	
6- Particles	
6.1- When an element slides on an iceberg, it will release white dust	1
6.2- When an element falls into the water, it will splash ellipsoid particles of water	2
7- More Game Elements	
7.1- Penguins of different colors, each with a different speed. In addition, it will be added penguins with negative velocity: they will move in the opposite direction to the inclination of the platform	4
7.2- Cubes to push with a larger size, implemented so that they are impossible to move if they are not being pushed by a specific number of penguins at the same time	4
7.3- Bomb cubes: Model bombs into ice cubes. Implement them so that when a penguin touches them, explode and the penguin comes out propelled. Manually program this reaction, without using the physics engine. When the bomb explodes, it expels particles of sparks	6
7.4- Bomb penguins: Penguins that, when you touch them, swell up growing in size quickly, and deflating again. The penguin that touches it will be propelled as if it were a bomb	6
7.5- Extra cold mini icebergs: Modeling them. Make them emerge particles of cold fog. Implement that, when touched, these will unravel with an explosion of particles, and the penguin that touches it will be frozen in motion for a short period	6
8- Sound	
8.1- Find (or record), edit and set sound effects on the interactions of elements with the environment	3
8.2- Find information and solve the delay in sound reproduction produced by Unity3D on Android **[6]**	12
8.3- Find, edit and set royalty-free music and ambient sound waves, giving the player the choice of listening one or other	4
9- Interface	
9.1- Place an interface that allows you to numerically display how many elements you have left to throw out, enter the pause menu, deactivate or activate the music, restart, exit, etc. Add to Unity3D a new text font suitable to the setting.	6
10- Accelerometer	
10.1- Find information on how to integrate acceleration usage with Unity3D on Android	2

10.2- Implement accelerometer data handling to give an almost identical gyroscope experience in its possible absence	15
10.3- Detect whether the mobile device has a gyroscope. Otherwise, force the use of the accelerometer. If it has both, give the player the option to choose sensor	2
11- Levels Map	
11.1- Manually implement the motion of the camera on the levels menu by sliding the finger on the touch screen, with inertia of scrolling on release	3
11.2- Model and include small icebergs that will be the levels, and paths between them, all of this over water. A penguin will walk over them to the level you select. If he is far away, he will get into the water and go to him diving	5
11.3- Find, edit and include music without copyright	3
11.4- Interface with remaining lives, options menu, etc	5
11.5- Programming an encrypted local data storage system	8
11.6- Implement the possibility of "sharing" with social networks	12
12- Art 2D	
12.1- Design and include a cover for the beginning of the game	12
13- Levels Creation	
13.1- Modeling multiple platforms to use in different levels	8
13.2- Design and implement a high number of levels	22
14- Testing	
14.1- Check all functionalities and test all levels that make up the app, both throughout the creation of the project and in its entirety at the end	14
14.2- Solve all errors, conflicts and problems encountered, both throughout the creation of the project and in its entirety at the end	15
14.3- Improve the first version of all project functionalities and components	5
15- Project Defense	
15.1- Writing work memory	25
15.2- Design and creation of the game presentation video	3
15.2- Prepare, rehearse, and perform the jury presentation	8
Total	**300**

Taking into account this table, and considering the aim to dedicate approximately 20 hours per week to the project, the resulting weeks planning would result as follows:

Date	Tasks
Week 1	1
Week 2	2
Week 3	3, 4
Week 4	4
Week 5	5, 6, 7
Week 6	7
Week 7	8
Week 8	9, 10
Week 9	10, 11
Week 10	11
Week 11	12, 13
Week 12	13
Week 13	14
Week 14	14, 15
Week 15	15
July 2nd	Submit Project

As I said, this schedule is just provisional, an initial personal guide, and it will be compared with the finally followed schedule at the end of this document, once the project is finished.

3. Conceptual Design

This part of the document presents the Game Design Document (GDD) of the developed game, as learnt on the degree subject VJ1222 - Video Game Conceptual Design.

3.1 Game Goals

Penguin Battle Royal is a 3D puzzle game for mobile Android devices, landscape orientation. The game is aimed to players of all ages and both genres. It is about penguins who are in the north pole. One of those penguins, the blue one, is the protagonist of the game. Player must rotate its game device to rotate the iceberg over which the penguins and items are. The main objective of each level is to throw everything over the platform to the water but the blue penguin. It is a single-player game, although players can obtain rewards by inviting friends to play.

3.2 Game Controls

Map of Levels Screen Controls:

- Sigle touch: Select a level, or any of the available interface options.
- Sliding touch: Scroll the camera over the map of levels.

In-Level Controls:

- Rotate the device: Rotate game's platforms (icebergs)
- Single touch: Recalibrate platform's original position, or any of the available interface options.

3.3 Technological Requirements

Target specs of the android device needed to run the game are the availability of one of these controllers:

- Gyroscope: Preferably.
- Accelerometer: More common than gyroscope. Accelerometer will emulate gyroscope behavior at its absence.

The game will work with a low number of Draw Calls [4], so android devices with low-capacity RAM will be able to run it.

Tools to use for the development of the game are: Unity3D, Monodevelop, 3D Studio Max, Adobe Photoshop, and Audacity, covering the manageability of each game component.

Regarding to cheat protection, the game encrypts its save data, and clock time is retrieved from internet instead of from the local device.

3.4 Title Screen

First thing player sees once the game is started is the title screen. It will be filled a 2D static image representing the game spirit, with a sense of dynamism. This image is not a screenshot of the game, but an artistic picture drawn with photoshop.

The title screen appears accompanied with its own audio song, which will not be listened at any other part of the game.

Once the player touch any part of the screen, he or she will be redirected to the Map of Levels Screen. In the case that player's device has neither gyroscope nor accelerometer, it is at this point when he or she will me warned that the game can not be player in that device.

3.5 Map of Levels Screen

This screen is the headquarters of the player. From here, he or she can edit any game option, check his or her records, enter in the shop, read every help note appeared so far, and select which level is going to play.

This screen shows a 3D scene: water, tiny icebergs with numbers (levels), the blue penguin standing over one those icebergs, and paths between the unlocked levels. On the top of the screen, an interface bar contents some player information, and also buttons to enter the options screen, shop screen, or help screen.

As happened with the Title Screen, this screen also appears accompanied with its own audio song, which will not be listened at any other part of the game. Besides, sound effects are played when player touches a level, when the blue penguin is walking and when the blue penguin splashes at entering or exiting water for diving.

Available options to be set once player touches the "options" button are the following:

- Mute / Enable every game sound
- Mute / Enable map of levels music
- Mute / Enable map of levels sound effects
- Select between music, waves sound or mute for levels
- Mute / Enable levels sound effects

- Advanced Options:
    - Enable / Disable shadows (for performance)
    - Select controller to use: Gyroscope (recommended) or Accelerometer

Every of these game options preferences will be saved in memory, so they will keep its state selected by the player between games. In the case of controller selection, it will be forced to accelerometer in the case that the player's device has not gyroscope.

Levels are distributed on horizontal rows of five levels each one. When player touches a level, it can happen different things depending on the number of rows of distance from the blue penguin to the selected level:

- Same row or just 1 row of distance: The blue penguin will walk to the selected level over the paths between levels.
- 2 or more rows of distance: The blue penguin will jump to the water and will dive quickly to the selected level. Paths between level will move if they represent an obstacle for the penguin when he is jumping to/from the water, and they will later recover their original position.
- The same level that was previously selected: The Level Info Screen will appear. If the blue penguin was still walking or diving to this level, he will instantly teleport to it.
- A level that is not unlocked: Nothing will happen. Locked levels can not even be selected.

The Level Info Screen of a specific level owns the following elements:

- The level number
- The three stars than can be obtained in that level. An obtained star will be filled in yellow. A not achieved star will be empty in color, and it will show the maximum level time required to obtain that star. In order to obtain all the three stars, player needs to finish that level really quickly.
- The "play" button that will lead the player to play that level.

At any time, the "back" button of the android device can be pressed. In that case:

- If the current screen is a sub-screen of the Map of Levels, game will come back to the Map of Levels.
- If the current screen is the Map of Levels, game will ask player if he or she wants to close the application. If so, the game will safely close.

3.6 Game Flowchart

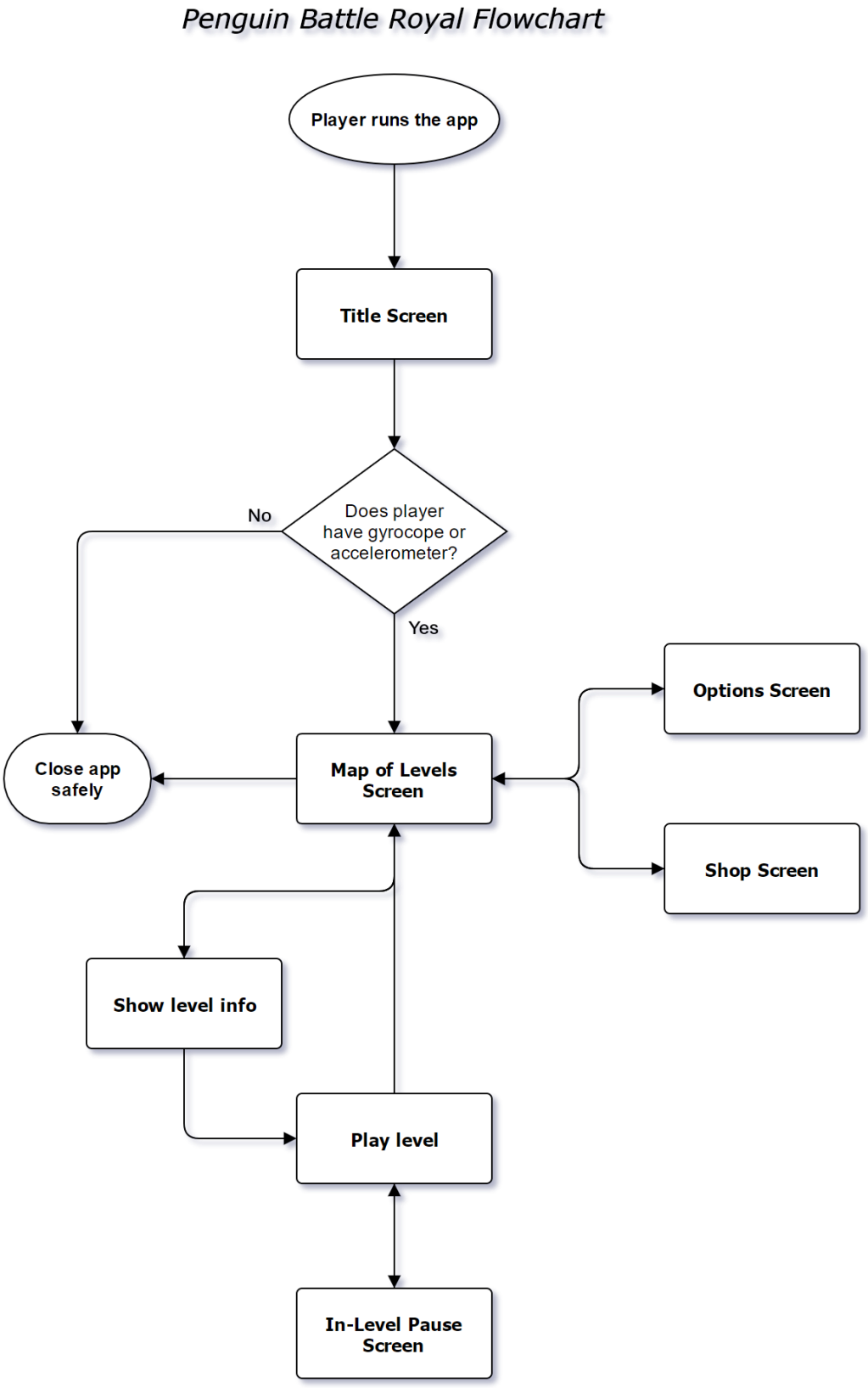

Fig. 1 - Game Fowchart

3.7 Loading Screen

Every time a new screen is going to be loaded, in the case that the loading process could take more than 1 second, the loading screen will be shown. It just consists on showing a “LOADING…” text over the current screen being paused. As soon as the new scene is loaded, the Loading Screen will disappear.

3.8 Game Camera

There are two different game cameras in the game:

- Map of levels camera: This camera is orthographic and third person. It can scroll through its Z axis by sliding a finger over the screen. When player releases the finger after a quick sliding, the camera will continue moving due to its inertia, and it will continuously slow down until finally stop.
- In-Levels camera: This camera is perspective and third person. It can not be moved by any event, neither by game mechanics nor by player inputs.

3.9 HUD System

Map of Levels Screen HUD: It is located at the top of the screen, with the simplistic shape of a rectangle filled with the following HUD elements:

- Total number of achieved Starts
- Number of Red Hearths
- Number of Blue Hearts
- Options Screen Button
- Shop Screen Button
- Help Screen Button

On-level-selected HUD: Once you select a level of the Map of Levels, an info screen with the following elements will appear:

- Level Number
- Achieved Stars
- Time needed to obtain not achieved Stars
- Play Level Button

In-Levels HUD: It will be as simple as possible for not filling too much of the screen.

- Number of resting elements to throw out of the iceberg and win (corner up left)
- Pause Menu Button (cornet up right)

On-Paused-Level HUD: A panel will appear in the center of the screen with the following components:

- Button to select between music, waves sound or mute
- Button to mute / enable sound effects
- Resume Button
- Exit Button (go back to the Map of Levels)

3.10 Player Character

Player character is a light blue penguin. There are many penguins in the game with different colors, all of them repeated, but player's penguin is the only penguin of the game that is light blue. That penguin has a normal velocity on platforms, and doesn't have any special ability. Even with that, player must ensure its survival over the platform while trying to throw the rest of penguins and items away to win.

3.11 Game Mechanics

When playing a level, many penguins and items may be over an iceberg platform. They move depending on the platform inclination, and depending on interactions between them. Game mechanics are the following:

- Rotate the iceberg platform: Player can rotate its physical device in order to smoothly rotate the iceberg platform in the same direction. Penguins over the platform will move towards the lower part of the platform, as seems logical in real life. However, their movement speed will depend on their color, as penguins of the same color have the same speed while penguins of different color have different speed. There are even penguins of negative speed, and their movement is towards the upper part of the platform, which will challenge player logics. On the other hand, regarding to items over platform such as ice cubes, they will not move due to this mechanic, no matter whatever inclination the iceberg platform has.
- Recalibrate the iceberg rotation: When touching on the screen, platform will set itself completely horizontal. We will name this mechanic "recalibration". Player may want to move his or her body without influencing the game, so the

way to do that is as simple as performing the body movement and then touching on the screen to recover the horizontality of the iceberg platform.

- Interaction between on-platform elements: Interactions are different depending on the elements that interact.
    - Pushing: Penguins are in movement, so their kinetic force may push other elements such as penguins or even items, which don't have other way to move. Heavy items may need to be pushed by more than one penguin to be moved.
    - Explosion: Specific elements of platform may explode when touched by another element. That another element will receive a strong force in the direction of the vector that comes from the exploding element to this one.
    - Freezing: Specific elements of platform may freeze penguins if they get in touch. A frozen penguin will not be affected by the platform rotation, by pushes or by explosions. A frozen penguin will be completely immovable until it recovers it standard state some seconds later.

3.12 Game Challenges

The game is intended to challenge both physical and mental abilities of the player:

- Physical coordination challenges:
    - Speed and reaction time: Winning is not the only objective of levels, but also obtaining all the three starts by winning before a specified limit time. When trying to win quickly, it becomes difficult to notice every interaction between on-platforms elements, so player must challenge his or her reaction time and act consequently to win at high speed.
    - Accuracy and precision: Player controls the game by using the rotation of his or her device. That rotation is recognized with an amazing amount of angle values, so every tenth of a degree counts. Player must rotate his or her device very lightly when needed, or heavily to win quickly, always towards the precise wanted direction.
    - Intuitive understanding of physics: The game has its own physical properties, and player should understand them to predict situations. Some examples are the strength of the explosive interactions, heavy items that need to be pushes by more than one penguin, etc.
- Mental challenges: After all, this game belongs to the puzzle genre. Every level involves a different combination of the elements that player needs to manage. In each level, player should observe the platform shape, which

elements are over that platform, and how are they distributed. Taking that information into account, it comes the time to elaborate a strategy, to think which elements is he or she going to throw away first and which later in order to win without throwing the blue penguin away.

On the other hand, the game has a clear structure of challenges:

- Main goals: Player may have two main goals depending on his or her playing style. These are unlocking and winning as many levels as possible, and obtaining as many stars as possible. Each level offers a maximum of three stars, so the first goal is need to fulfill the second one. So, although the first goal is mandatory and the second one is optional, some players may care more about the first one and some others about the second one.
- Main goal: Unlocking and winning as many levels as possible.
- Secondary goal: Obtaining as many stars as possible. These are obtained by winning levels before a specified limit time. Each level offers a maximum of three stars, so the main goal is needed to fulfill this one.
- Levels victory conditions: A level may start with penguins and items over a platform. To win, player must throw everything away from the platform but the player character, a blue penguin. If the blue penguin falls away, player loses.
- Atomic challenges: To accomplish the levels victory conditions, player must manage every interaction between on-platform elements, for example trying to save the blue penguin when it receives the force of an explosion. Several atomic challenges may appear simultaneously.
- Implicit challenges: Player is free to choose how to overcome a level, but the design of the game offers the player a more logical way to do it. For example, due to the game design, player should explode bombs by touching them with other penguins as it doesn't matter if they fall away, but he or she may want to explode them by touching them with the blue penguins, after what it will be very hard (although not impossible) to save it to be thrown away.
- Explicit challenges: Each level shows clearly the time limit needed to obtain its stars. In the case that player wanted those, it is explicitly described the challenge to overcome to obtain them.

3.13 Game Rules

- -Foundational rules (inside game's logic):
    - When two elements get in touch, they interact consequently.
    - Once an element gets outside the iceberg platform, player loses any control over it. If it is a penguin, it will swim away. If it is an item, it will sink to the sea bottom.

    - An element outside the platform has not any way to come back onto it.
    - When player throws away everything but the blue penguin, he or she will still need to wait some seconds without throwing away the blue penguin to win.
    - Penguins' speed will depend on the platform's inclination, their own speed parameter (depending on their color), and the interaction between them.

- Basic/Operational rules (inside player's mind): There are so many as limited strategies to win in each level.
    - If I have the blue penguin, two normal penguins, and a cube that needs to be pushed by at least three penguins, I will need to throw away the cube first or I will not be able to do it later with fewer penguins.
    - If I have the blue penguin, a bomb, and a wall, I will need to touch the bomb with the blue penguin in the right direction to be later stopped thanks to the wall, or I will lose whatever I try.
- Written rules: All of them will appear on-screen, and they will be able to be revised acceding to the help screen.
    - Touch a level to select it and play it.
    - Rotate the device to rotate the iceberg platform.
    - Touch the screen to recalibrate the iceberg platform.
    - Overcome a level to unlock the next one.
    - (Rules of every new on-platform element that may appear)
    - Select a level that is two or more rows away to go there diving.

3.14 Game Balancing

Penguins Battle Royal is a PvE game. Its relationships among the player's options are largely intransitive, since player's character has not any kind of upgrades or power ups. Its relationships depend on the characteristics of each level and the amount of possible strategies that player could perform to accomplish it. The way in which the game maintains the flow channel is by continuously raising the levels difficulty while inserting new kind of on-platform elements.

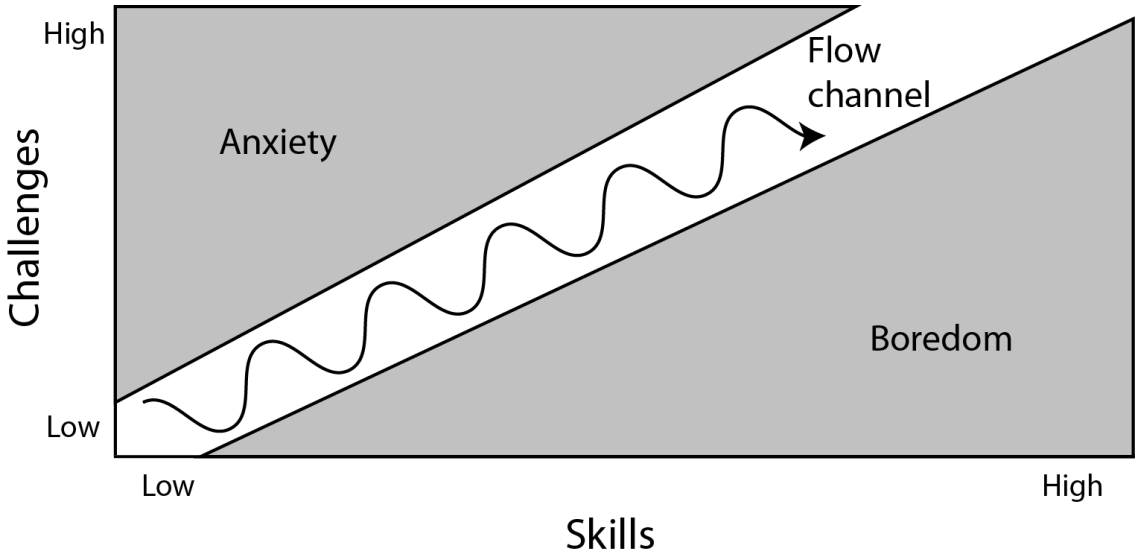

Fig. 2 - Desired Flow Channel for a Balanced Game

The idea is that player never perceive to be unfair any element of the game. Everything that happens in the levels is only because of player actions and because of understandable game physics.

Game difficulty increases with few troughs: the insertion of new elements. Each time a new element is presented to the player, the few following levels will easier to serve as a tutorial for the new included characteristics. On the other hand, the game difficulty has specific peaks: the special levels. Each time player overcomes a level multiple of ten, a special level will be unlocked. Special levels are not required to be played or overcome, but they are hard, fun and a way to obtain more stars.

3.15 Scoring

The Scoring Points that this game owns are the called “Stars”. A player has a total number of obtained stars, and a number of obtained stars in each level, from 0 to 3. As explained before, stars are rewards given for accomplishing a level before a specified limit time. There are 3 obtainable stars per level, each of them with a different required time limit, from easier to harder.

3.16 Game Elements

There are two main kind of elements that may be over the levels’ platform:

- Penguins: Any of them can be called “enemy” as they don’t really attack you. They can not be called NPCs either, as they are not controlled by a IA but by player’s actions. All of them move depending on the platform rotation, and when they fall away to the water, they swim away. There are several types of penguins:

    - Light Blue Penguin: This one is the Player's Character. It is described at chapter "3.10 Player Character". There are only one Light Blue Penguin in the entire game, and of course it appears on every level.
    - Dark Blue Penguins: They have exactly the same velocity than the Light Blue Penguin.
    - Yellow Penguins: They are faster than the Light Blue Penguin.
    - Red Penguins: They are even faster than the Yellow Penguins.
    - Pink Penguins: They are slower than the Light Blue Penguin.
    - Orange Penguins: They are even slower than the Pink Penguins.
    - Green Penguins: They have negative velocity. What this means is that they move in the opposite direction than the rest of the penguins, challenging laws of gravity,
    - Black Penguins: They are also called Bomb Penguins. When they get in touch with another penguin, they get nervous and inflate their bodies as fast as produce the explosion interaction mechanic: while they inflate, every element that touches them will receive an orthogonal force, acting this Black Penguin as a bomb. After they quickly inflate, they also quickly deflate and recover they original position. They are as slow as Orange Penguins.

- Items: Any of them move whatever rotation the platform has. When they fall out to the water, the slowly sink. They also need to disappear from the platform to win, so player must move penguins toward them to get them away.
    - Ice Cubes: They are simple cubes than need to be pushed away by penguins.
    - Long Ice Cubes: They are heavier and need to be pushed away by more than one penguin. One characteristic than player need to keep in mind is than they need to be touched by their required number of penguins. It is not sufficient if several penguins are in line pushing a Long Ice Cube. They need to be touching it. This is the reason why they are "long". It is a required challenge to position penguins on the right position. As a strategic tool, they can also be moved by explosion interactions even with just one Black Penguin, although that Black Penguin will not explode if it is not touched by another penguin.
    - Ice Bomb Cubes: They are ice cubes that have a bomb inside. They can not be pushed but exploded. Any element should touch them to make them disappear, taking into account that the sacrificed element will receive a strong force.
    - Tiny Extra Cold Icebergs: As Ice Bomb Cubes, they can not be pushed but "destroyed". When a penguin touches them, they get destroyed and

freeze that penguin for some seconds, performing the freezing mechanic (a frozen penguin can not be moved by anything).

3.17 Game Levels

Game levels are numerous and short in time to accomplish, as happens with most of successful mobile games. They are composed by a platform (with different shape depending on the level) and some elements of the previous chapter over that platform. A level is overcome when the only element that remain over the platform is the Light Blue Penguin, the Player's Character. A level is failed if the Light Blue Penguin falls before overcoming the level.

Levels will start very easy and will slowly increase in difficulty. When a new element is introduced to the player and starts to appear in the levels, they reduce their difficulty to serve as a tutorial, letting the player to understand the new element.

Each time player overcomes a level multiple of ten, a special level will be unlocked. Special levels are not required to be played or overcome, but they are hard, fun and a way to obtain more stars. They usually have a higher number of on-platform element and a wider platform.

If the game success, new levels will be developed to avoid players leaving the game. If so, new elements will be also developed. However, this is matter of the future.

3.18 Monetization

The game will be downloadable for free in Google Play for android devices. Nevertheless, it will implement two different technologies to obtain money: Rewarded Video Ads [7] and IAP (In-app Purchases) [8].

There are two consumable elements that will compose the monetization system of the game:

- Red Hearts: Player starts the game with three Red Hearts. Every time player plays a level, he or she needs to spend one Red Heart. Player can recharge one Red Heart whenever he or she wants by watching a Rewarded Video Ad [7]. Every day, player wakes up with all the 3 Red Hearts as they are fully recharged at night.
- Blue Hearts: Every time player plays a special level (those that are unlocked every time player overcomes a level multiple of ten) he or she needs to spend one Blue Heart. Blue Hearts can be also spent to play usual levels when player has no Red Hearts, but it is not recommended since Blue Hearts are harder to obtain.

Blue Hearts are the tool to reach a viral marketing: Player has the option of "sharing" the game on social networks (whatsapp, facebook, twitter, etc). When doing so,

the shared message will be “Join Penguin Battle Royal! Play it now and insert my nick <nick> to start with two Blue Hearts! <download link>”. When player plays for first time, he or she will be offered to insert the nick of the person who may have recommended the game to him or her. If so, player will start with two Blue Hearts. Besides, when player shares the game for first time, he or she will be asked for a personal nick for sharing. Player who shares becomes also benefited because obtains eight Blue Hearts when people joined with his or her nick pass the level number ten (for avoiding fake sharing). Due to this fact, sharing is desired for both sides, so it is intended to become frequent habit.

We have explained the inclusion of Rewarded Video Ads and the viral marketing planning. Last but not least, the game will include a virtual shop with the next in-app products:

- 10 Blue Hearts: This product can be bought for approximately 1€.
- Golden Heart: It supposes infinite Red Hearts and no more video ads. This product can be bought for approximately 2€. It is intended to be the main source of monetization, as it is a lovely deal. People out of home, without wi-fi, may hate watching video ads due to their data spending, and this purchase would solve that issue.

3.19 Music and SFX

The game will content different pieces of music and sound effects. They will not be recorded specifically for the development of the game, but all of them will own free copyright for commercial purposes. For those pieces of music with Creative Commons Attribution (3.0) license will be a mention in the game to the author and online source, as needed for this kind of license.

The game needs three pieces of music for the following parts of the gameplay:

- Title Screen: It needs a startling song. That song should be self-sufficient to hold player in this simple screen just for listening the whole song. The chosen song is: *Dollheads by Ivan Chew (c) copyright 2010 Licensed under a Creative Commons Attribution (3.0) license.* http://dig.ccmixter.org/files/ramblinglibrarian/25202.
- Map of Levels Screen: It needs a loop song with some personality. That song should be also enough to hold player, but without stealing importance to the map of levels itself. The chosen song (after being modified to serve as a loop) is: *Drops of H2O (The Filtered Water Treatment) by J.Lang (c) copyright 2012 Licensed under a Creative Commons Attribution (3.0) license.* http://dig.ccmixter.org/files/djlang59/37792 *Ft: Airtone*.
- -In-Levels: Levels need an ambient loop music enjoyable but simple enough to avoid becoming boring or repetitive. There are two chosen music for this, and player will have the option to choose the one he or she prefers. One of them is the simple and relaxing sound of sea waves. The other one is the

following piece of music (after being modified to serve as a loop): *Chords For David by Pitx (c) copyright 2011 Licensed under a Creative Commons Attribution (3.0) license.* http://dig.ccmixter.org/files/Pitx/30638 *Ft: jlbrock.*

Regarding to sound effects, there will be several of them to serve at the following situations:

- When a button is pressed (a classic "pop" sound).
- When the penguin is walking (its footsteps)
- When something is splashing at the water.
- When a bomb explodes.
- When a tiny extra cold iceberg gets broken.
- When player overcomes a level.
- When player fails a level.

4. Functional and Technical Specifications

4.1 Gyroscope

"A gyroscope is a device that uses Earth's gravity to help determine orientation. Its design consists of a freely-rotating disk called a rotor, mounted onto a spinning axis in the center of a larger and more stable wheel. As the axis turns, the rotor remains stationary to indicate the central gravitational pull, and thus which way is down" [9].

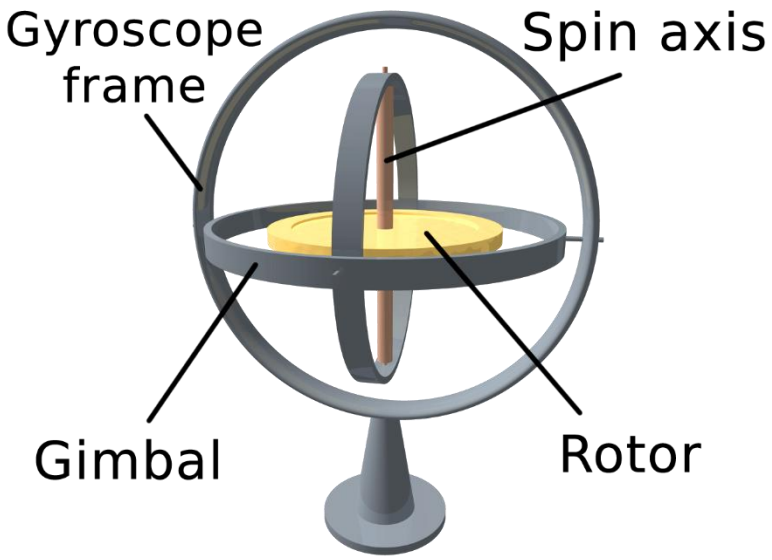

Fig. 3 - Gyroscope

Many Android devices own a gyroscope sensor. Its extreme precision about the device rotation with respect to the center of the Earth is the reason why it has been chosen for the control of the game platforms rotation.

Regarding to Unity3D potential, the game engine that powers the project, there are two input values that may be useful for our objective [10]:

- Rotation Rate: This input value consists on a Vector3 [11] representing the speed of rotation around each of the three axes in radians per second.
- Attitude: This input value consists on a Quaternion [12] representing the attitude (orientation in space) of the device.

A basic scenario was set to test the usability of both input values with respect to our objective. This scenario was composed by a wide and short cylinder, which could be a simple and nice platform for some levels. We seek for the rotation of that platform relative to the rotation of the android device in real time.

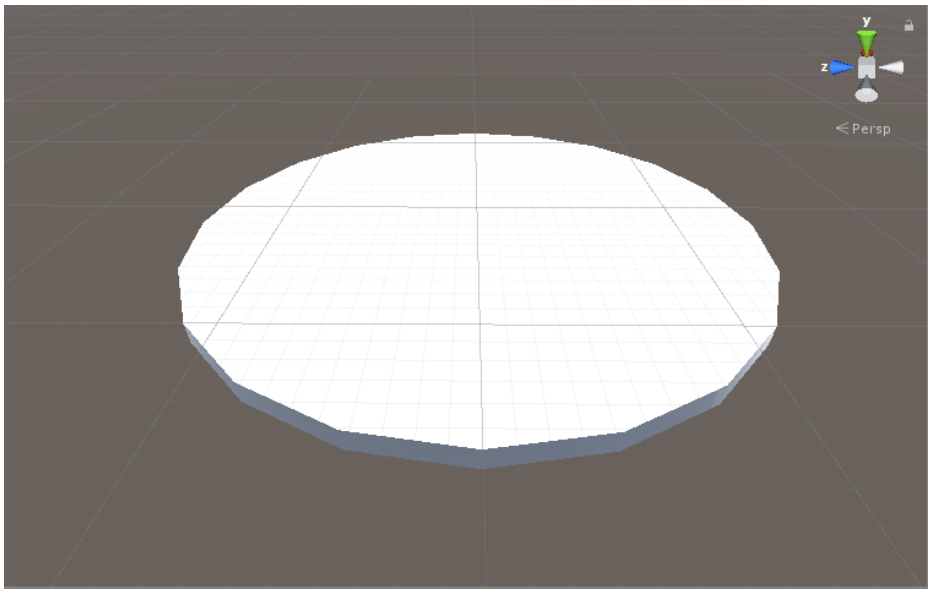

Fig. 4 - Simple Platform to be rotated with android device gyroscope

We need to keep in mind which axes we want to rotate around: looking at the previous figure, we want the platform to rotate freely around the Z axis (blue arrow) and X axis (red arrow), axes that will change the inclination of the platform (as Y axis would not), which will actually displace items on platform in the future game.

As a first attempt, the Rotation Rate input value was used to reach the desired behavior. The implementation code is shown next:

```
public Transform platform;
private float platformSpeed = 0.7f; // 1 is real speed
private Vector3 previousRotation;


void Awake(){
    // Activate the gyroscope
    Input.gyro.enabled = true;
    previousRotation = new Vector3 (0,0,0);
}

void FixedUpdate () {
    float newX = previousRotation.x - Input.gyro.rotationRate.x
* platformSpeed;
    float newZ = previousRotation.z - Input.gyro.rotationRate.y
* platformSpeed;
    previousRotation = new Vector3 (newX, 0, newZ);
    Vector3 newRotation = new Vector3 (eulerUntil180(newX), 0,
eulerUntil180(newZ));
    platform.eulerAngles = newRotation;
}

float eulerUntil180(float rot){
    //Transform to [-180, 180]
    if (Mathf.Abs (rot) > 180){
```

```
            if (rot > 0) while (rot > 180) rot -= 360;
            else while (rot < -180) rot += 360;
        }
        return rot;
    }
```

As it can be seen, the first thing that gyroscope needs to work is to enable it with the command “Input.gyro.enabled = true”. This will not be need with the accelerometer, as we will notice in the next chapter of this technical report.

By using Rotation Rate, we obtain a relative change from the previous state, so we will be working with two values: “previousRotation” and “newRotation”. Now, we have to modify the “previousRotation” to become “newRotation”, and we do so by modifying de X and Z euler angles (axes that we need, as seen before) in accordance with the Rotation Rate values. Trying different combinations, the achieved modification to obtain the desired behavior is to modify the X angle with the X input, and the Z angle with the Y input. As we are working with Vector3, we need then to make sure than the angle does not go further than a frustum of 360 degrees to avoid unexpected behaviors when manually modifying the angles.

The final result seems nice. Behavior looks exactly as desired. The objective seems reached, but it is then when we try to play it with our body looking at a different position, for example looking at north if we were playing looking at south. This way, rotation is inverted, so this solution should take into account some physical states to finally work. For this reason, the Attitude input would be checked before going any further with this method.

As a second attempt, the Attitude input value was used to reach the desired behavior. The implementation code is shown next:

```
public Transform platform;
private float platformSpeed = 0.3f; //1 is real speed
private Quaternion initialRotation;

void Awake(){
    //Activate the gyroscope
    Input.gyro.enabled = true;
    initialRotation = Input.gyro.attitude;
}

void FixedUpdate () {
    Quaternion relativeRotation = Quaternion.Inverse
(initialRotation) * Input.gyro.attitude;
    Vector3 currentRotation = relativeRotation.eulerAngles;
    Vector3 fixedRotation = new Vector3 (eulerUntil180 (-
currentRotation.x), 0, eulerUntil180 (-currentRotation.y)) *
platformSpeed;
    Vector3 oldEulerAngles = platform.eulerAngles;
    platform.eulerAngles = fixedRotation;
}

//Transform to [-180, 180]
public static float eulerUntil180(float angle){
```

```
    if (Mathf.Abs (angle) > 180){
        if (angle > 0) while (angle > 180) angle -= 360;
        else while (angle < -180) angle += 360;
    }
    return angle;
}
```

This version obtains the relative rotation operation with quaternions, with are more accurate and engine-friendly when working with rotations. We need to save the initial rotation attitude of the android device to rotate the platform relatively to that position. From now on, we will set the platform rotation comparing the initial device rotation with it current rotation. This process is different to the previous method, where we modified the rotation relatively to its closely previous state.

The relative rotation is obtained by multiplying the inverse quaternion of the initial state by the current rotation state. Once we obtain the relative rotation, we apply it to the platform, but only at the two desired axes (X and Z). As before, after trying different combinations, the achieved modification to obtain the desired behavior is to modify the X angle with the X input, and the Z angle with the Y input, and then we make sure that those angle values do not go further our frustum.

Result seems nice again. Furthermore, this time it works with any initial angle that could be used. For this reason, next step is to improve this method as to achieve next objectives.

We want to be able to “recalibrate” the platform rotation when desired by simply touching the screen. This means, instantly recover a horizontal rotation, and then consider that position of the android device as the new initial state to rotate relatively to. We do it this way:

```
void FixedUpdate () {
    if (Input.touchCount > 0) {
        Recalibrate ();
    }
    ...
}

public void Recalibrate(){
    initialRotation = Input.gyro.attitude;
}
```

Next objective is to limit the rotation of the platform to an appropriate degree of inclination trying to get closer to the game idea. We limit its rotation this way:

```
private int maxInclination = 5;

void FixedUpdate () {
    ...
    ...
```

```
    Vector3 limitedRotation = new
Vector3(limitRot(fixedRotation.x), 0,
limitRot(fixedRotation.z));
    ...
    plataforma.eulerAngles = limitedRotation;
}

float limitRot(float rot){
    if (Mathf.Abs (rot) <= maxInclination)
        return rot;

    if (rot > 0)
        return maxInclination;

    return -maxInclination;
}
```

Finally, we obtain exactly the desired behavior of the platform when rotating the device. How we will use that rotation for moving on-platform elements will be treated in coming chapters.

4.2 Accelerometer

"An accelerometer is a compact device designed to measure non-gravitational acceleration. When the object it's integrated into goes from a standstill to any velocity, the accelerometer is designed to respond to the vibrations associated with such movement. It uses microscopic crystals that go under stress when vibrations occur, and from that stress a voltage is generated to create a reading on any acceleration" [9].

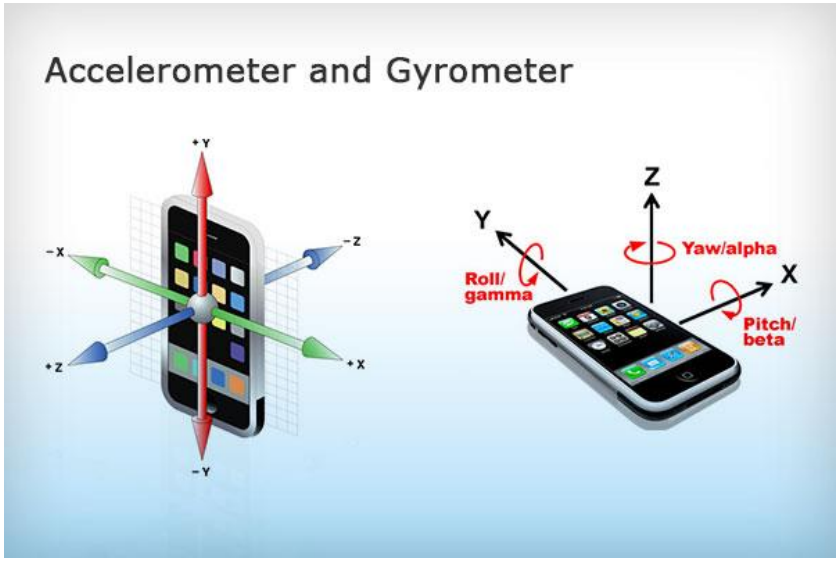

Fig. 5 - Accelerometer (left) and Gyroscope (right) comparison

It is such a challenge to emulate gyroscope behavior by using just accelerometer inputs, but it may be very useful since "most Android-powered devices have an accelerometer, and [just] many now include a gyroscope" [5]. This game is intended to reach as much players as possible.

First, let's understand which facilities Unity gives for obtaining accelerometer inputs [13]:

- Acceleration: This input value consists on a Vector3 [11] composed by the three coordinates of the last measured linear acceleration of the device in three-dimensional space.

Before trying to imagine how this input looks like, let's check it in practice:

- When moving the device in some direction, the magnitude of this Vector3 increments. The faster the device is moved, the larger the magnitude of the Vector3 gets. Coordinates of the Vector3 depend on the direction the device is being moved.

- When not moving the device at all, the magnitude of this Vector3 stay closer to one. In other words: With no displacement, the Acceleration input given by looks like a unit vector.

- When not displacing the device but rotating it (the expected behavior for our game), the unit vector described above changes its coordinates depending on which axis is the device being rotated around. This sounds interesting, but the Acceleration values that change depend on the device orientation. In other words: when rolling, pitching or yawing the device, the coordinates of the Acceleration input that will change will depend on it the device orientation, while the gyroscope input coordinates that change are always the same depending of the rotation direction (rolling, pitching or yawing).

Once knowing how this input works, let's try to approach our target: obtaining a similar control experience to the obtained with our gyroscope usage implementation. Next is the figured-out code that achieves it, trying to use the same coding structure that the used for gyroscope.

```
private bool gyroscope = false; //true for gyroscope, false for
accelerometer

public Transform platform;

private float platformSpeed = 0.3f;
private int maxInclination = 5;

private Quaternion initialRotation;


void Awake(){
    Recalibrate();
}

void FixedUpdate () {

    if (Input.touchCount > 0) {
        Recalibrate ();
    }


    Quaternion relativeRotation = Quaternion.Inverse
(initialRotation) * CurrentDeviceRotation();

    Vector3 currentRotation = relativeRotation.eulerAngles;

    Vector3 fixedRotation;
    if (gyroscope) {
        fixedRotation = new Vector3 (eulerUntil180 (-
currentRotation.x), 0, eulerUntil180 (-currentRotation.y)) *
platformSpeed;
    }
    else { // if accelerometer
        fixedRotation = new Vector3 (eulerUntil180 (-
currentRotation.y), 0, eulerUntil180 (currentRotation.x)) *
platformSpeed;
```

```
        fixedRotation *= 0.33f; // Trial and error value
obtained to get a rotation speed almost identical to the
obtained with accelerometer
    }

    Vector3 limitedRotation = new
Vector3(limitRot(fixedRotation.x), 0,
limitRot(fixedRotation.z));

    Vector3 oldEulerAngles = platform.eulerAngles;

    platform.eulerAngles = limitedRotation;
}

//Transform to[-180, 180]
public static float eulerUntil180(float angle){
    ...
}

// Limit an angle to maxInclination
float limitRot(float rot){
    ...
}

public void Recalibrate(){
    initialRotation = CurrentDeviceRotation();
}

private Quaternion CurrentDeviceRotation(){
    if (gyroscope)
        return Input.gyro.attitude;

    //If accelerometer
    Vector3 acceleration = Input.acceleration.normalized;
    if (acceleration.z > 0)
        acceleration *= -1; // to being able to play upside down

    Quaternion result = Quaternion.Euler(acceleration*180);
    return result;
}
```

This is the best approach that could be made. With this implementation, behavior looks perfect when the device initial position is parallel to the floor or closer, as it uses the right coordinates for this case. As mentioned above, when dealing with this acceleration input the coordinates to check change depending on the device attitude. For this reason, it looks hard to obtain a better behavior. It will be left for future work. Nevertheless, it is worth to say that playing with the device more or less parallel to floor (which is the most natural way to play this game) it works exactly as nice as it does with gyroscope usage.

Let's explain the code: As it can be read, now the device rotation is not obtained through "Input.gyro.attitude", but through a function that will return a value depending on the sensor chosen to control the game. This function, "CurrentDeviceRotation()", should return a Quaternion to ease compatibility. The way it is implemented to accelerometer is:

- Obtain accelerometer input value in Vector3 state.
- Normalize it to deal with players who are not just rotating but also moving the device.
- When parallel to floor, the Z axis of the accelerometer input gets negative. We check it to then multiply the whole Vector3 per -1 and being also able to play also upside down (this is just an additional detail, as almost nobody would play this way).
- Multiply the unit vector per 180, which is the maximum negative or positive value we want for any coordinate.
- Convert to Quaternion to get the relative rotation in "FixedUpdate" as it would do with gyroscope.
- One obtained the relative rotation, select the right vector3 coordinates we need, which are different to the ones obtained with gyroscope input.
- Multiply the resulting Vector3 per a Trial and error value obtained to get a rotation speed almost identical to the obtained with accelerometer.

Now, the platform rotates as it would do with gyroscope, but if we look in depth, we see that it seems the platform is "vibrating". This is because the floating and not fully precise data given by the gyroscope. This behavior is really annoying, so let's fix it with a Low Pass Filter, which filter high frequency jitter caused by hardware sampling such as accelerometer output or Augmented reality marker pose output [14].

```
private Quaternion CurrentDeviceRotation(){

    ...

    Quaternion result =
Quaternion.Euler(LowPassFilter(acceleration*180));
    return result;
}


private Vector3 previousVectorLowPassFilter;


private Vector3 LowPassFilter(Vector3 vector){
    if (previousVectorLowPassFilter == new Vector3()) {
        previousVectorLowPassFilter = vector;
    }

    float lowPassFilterValue = 0.25f;

    Vector3 result = Vector3.Lerp (previousVectorLowPassFilter,
vector, lowPassFilterValue);

    previousVectorLowPassFilter = result;

    return result;
}
```

The result is exactly the one we were looking for. “LowPassFilter()” lerps [15] (linearly interpolates) the previous rotation of the platform with the new one. Now it does not vibrate. It looks like player would be using gyroscope instead. The only noticeable difference is that this implementation of accelerometer usage does not support device attitudes far to the one parallel to the floor, so we must alert to the player in-game when he or she is approaching a attitude that my result in unexpected behaviors:

```
public GameObject accelerometerWrongInclinationAlert;

...

private Quaternion CurrentDeviceRotation(){
      ...
      accelerometerWrongInclinationAlert.SetActive(Mathf.Abs(acce
leration.z) < 0.5f);
      ...
}
```

Objective achieved: Being able to play this game with accelerometer emulating as much as possible gyroscope behavior.

4.3 Modelling and Animating

Game characters are going to be penguins, so it is need to model them. The chosen tool for modelling is 3DS Max, as it is the one that we have learnt to use in the degree subject VJ1216 - 3D Design.

The desired style is short and cute penguins. They must be short because in the game they must behave as a ball would, so their body should try to approach a spherical shape. The final result is the following:

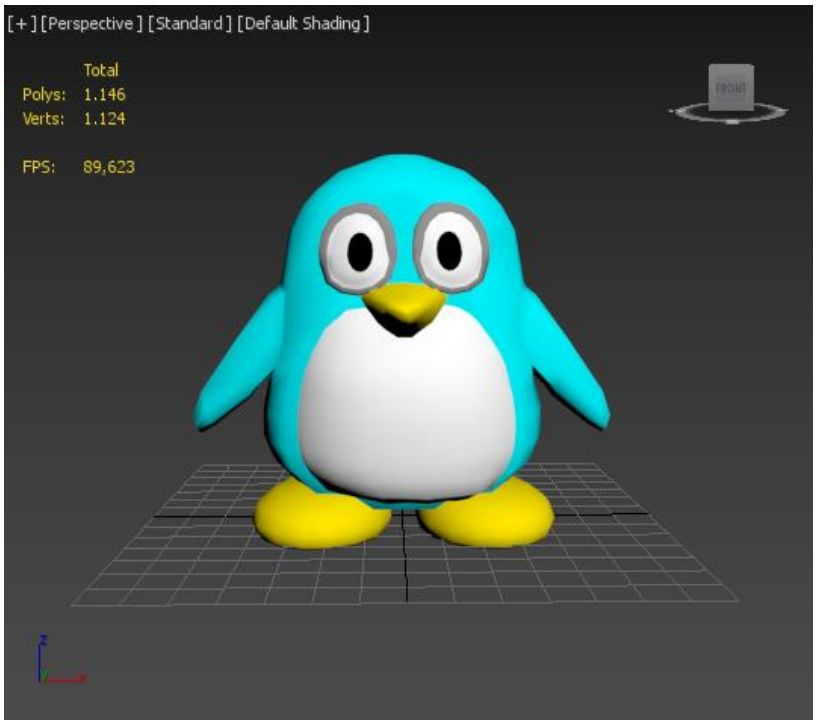

Fig. 6 - Penguin modelled with 3DS Max

Now that we have modelled or penguin as an editable poly, we need next to animate it as learnt in the degree subject VJ1226 Character Design and Animation. For doing that, we need first to add biped bones to it:

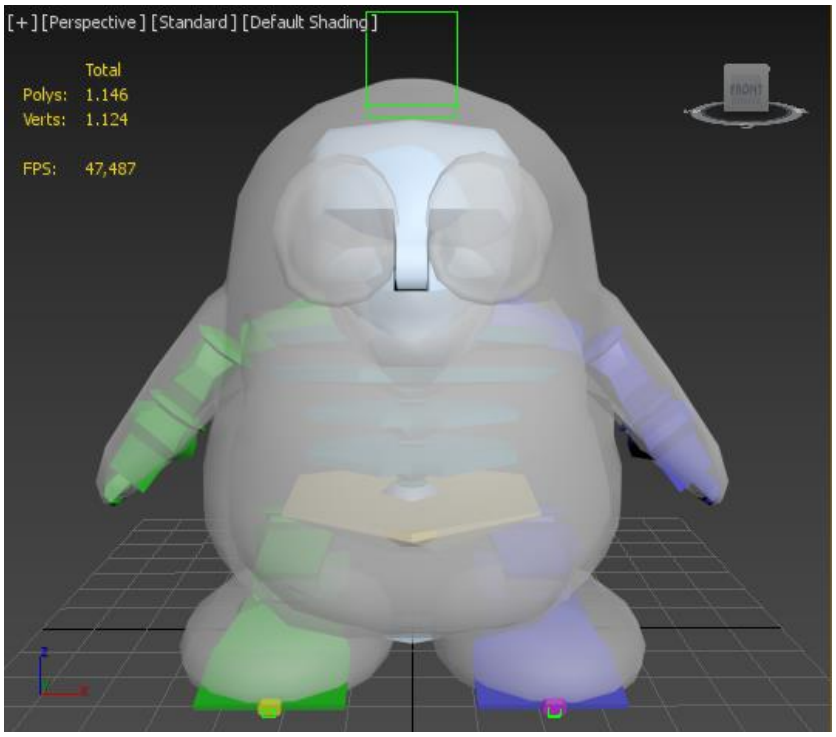

Fig. 7 - Biped bones for the model

If we want those bones to actually displace their respective part of the model when moved, we need next to adjust the envelopes of each link to select their influence area.

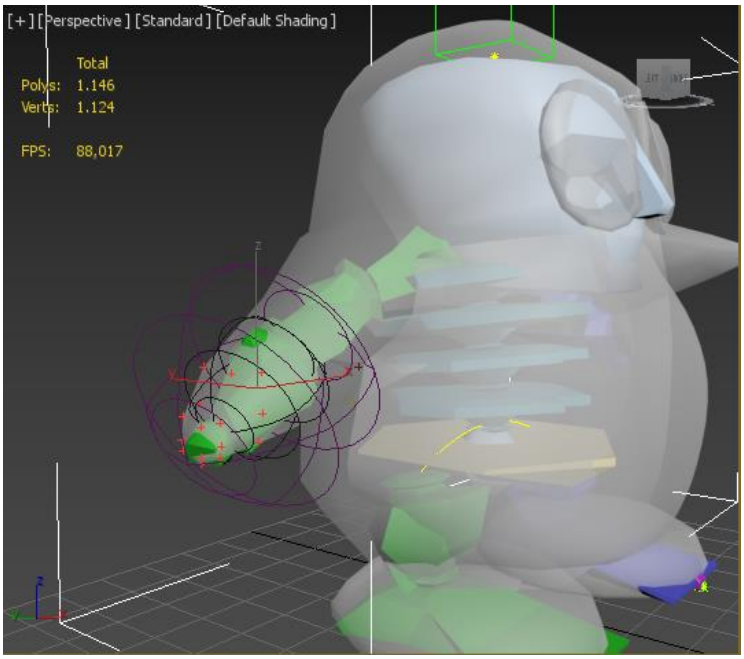

Fig. 8 - Biped envelopes for the right hand of the model

After adjusting every ingle envelope of the penguin, we can now move its bones and correctly displace the influence parts of the body for each bone. We can now those animations that are going to be needed for the game. These are: Idle, walking, jumping to get lying down, lying down idle, and swimming. These animations are created interpolating keyframes between different biped bones positions.

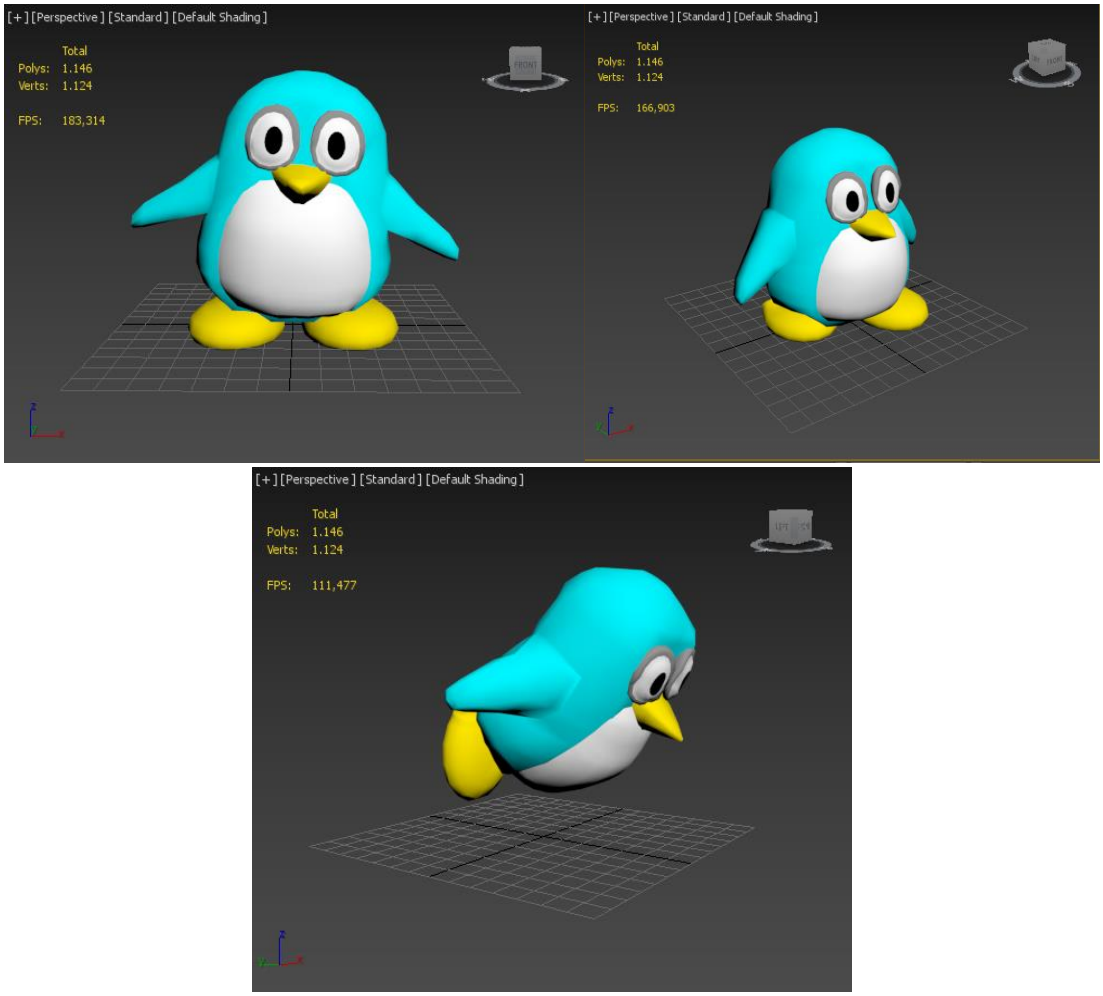

Fig. 9 – Biped animations for idle, walking, and jumping to get lyind down

For exporting the model with its animations for being usable with Unity3D, we first mix all biped animations inside a long frames list, one after the another. Then, we export it as a FBX object, and then import it inside Unity3D. Finally, in order to obtain the animations, we need to manually set the initial and final keyframe number for each one into Unity3D, name them, and create an animator component that can manage the transition between them in-game. One this is done, we have finished with our penguin design task.

4.4 Characters Behavior

Now that we have the model with animations for our penguins imported into the project, let's give our penguins functionality. However, we should first adapt the scene to look like a nice place for our penguins. It has been done by giving to the platform an iceberg texture, and positioning it over refractive water as offered by Unity3D [16]:

Fig. 10 - Penguins, water and iceberg. First nice looking scene.

What we want is penguins lying down sliding over the iceberg depending of its inclination, given by player control. We also want them to realistically turn towards the direction they are moving. Besides, we want they fall to the water when they get out of the platform. Once they reach the water, we want they go out of the screen by swimming in the right direction. Last but not least, we want to make it more realistic by providing a water splash particle system when they fall to the water, and a snow dust particle system when they are moving over the platform.

It was not intended to use the gravity provided by the unity physics engine and the inclination of the platform for the movement of the penguins, due to its imprecision, unpredictability, and the impossibility to implement that way the behavior of the green penguin (see chapter 3.16, Game Elements) and those item that only move if they are pushed. However, it was given a try because of its simplicity. Result was indeed unsatisfactory, as penguins become uncontrollable because of their mix of levitating and friction when platform rotates, and they tend to get closer each other even when they all have the same physical attributes. Therefore, let's continue with the initial plan of manually controlling their behavior.

Our plan is the following: Moving them in a phantom horizontal plane, and real-time position them in at the right height over the platform to avoid friction. They direction

at which they are going to move will depend on a custom gravity whose values are going to change at the same time the platform rotation does.

```
...
public static Vector3 myOwnGravity;

void FixedUpdate () {
    ...
    float gravityX = -platform.eulerAngles.z;
    float gravityY = platform.eulerAngles.x;
    myOwnGravity = new Vector3 (gravityX, 0, gravityY);
}
```

By doing this, we obtain a global static custom gravity that can be accessed from any script, and which depends on the platform rotation. Let's now make penguins use it, executing the following by each penguin:

```
    public float speed = 50;
    private Rigidbody myRigidbody;

    void Awake () {
        myRigidbody = GetComponent<Rigidbody> ();
    }

    void FixedUpdate () {
        myRigidbody.AddForce (new Vector3
(Gyroscope.myOwnGravity.x * speed, 0, Gyroscope.myOwnGravity.z *
speed));
    }
```

Now that we have penguins horizontally moving, let's position them at the right height over the platform to avoid friction. This will be performed always just after rotate the platform, by the same script which does that. This way physics will not act between both processes to cause possible frictions. We will check the corresponding height for each penguin by performing ray casting [17].

```
private Transform[] onPlatformTransforms; // Array with the
transform of every penguin
private bool[] onPlatformIsFalling; // Array that keep in mind
which penguins are still over the platform and which fell out

public LayerMask layerMask; // Raycast will only check for the
platform, which has this layer

...

void Start(){
    GameObject[] onPlatformObjects =
GameObject.FindGameObjectsWithTag ("OnPlatform"); // Every
penguin will have this tag
```

```
    onPlatformTransforms = new
Transform[onPlatformObjects.Length];
    onPlatformIsFalling = new bool[onPlatformObjects.Length];

    for(int i=0; i < onPlatformTransforms.Length; i++){
        onPlatformTransforms[i] =
onPlatformObjects[i].GetComponent<Transform>();
        onPlatformIsFalling [i] = false; // At the beginning, no
one is falling
    }
}

void FixedUpdate () {
    ...
    // Here up the rotation of the platform was set
    ...
    ...
    for(int i=0; i < onPlatformTransforms.Length; i++){

        if (!onPlatformIsFalling [i]) {
            RaycastHit hit;

            // Throw a raycast from above to down, checking for
the platform
            Physics.Raycast (onPlatformTransforms[i].position +
platform.up * 10, -platform.up, out hit, 15, layerMask);

            if (hit.collider != null) {
                // Put the penguin at that position
                onPlatformTransforms[i].position = hit.point;
            }

            // If the raycast doesn't find the platform, it is
because penguin is already falling
            else {
                onPlatformTransforms
[i].GetComponent<Penguin>().SetFalling (); // TODO do something

                onPlatformIsFalling [i] = true;
            }

        }
    }
}
```

Code above if fully commented: it takes every penguin, throw a ray cast at its position at its position from up to down, check where the platform is, and put the penguin at that position. If the platform is not found, the script tells the penguin it is falling out. We will cover it later, as now we will continue improving the behavior over the platform.

What we want now is to set penguins up vector parallel to platform up vector, as for now penguin up vector doesn't change at all.

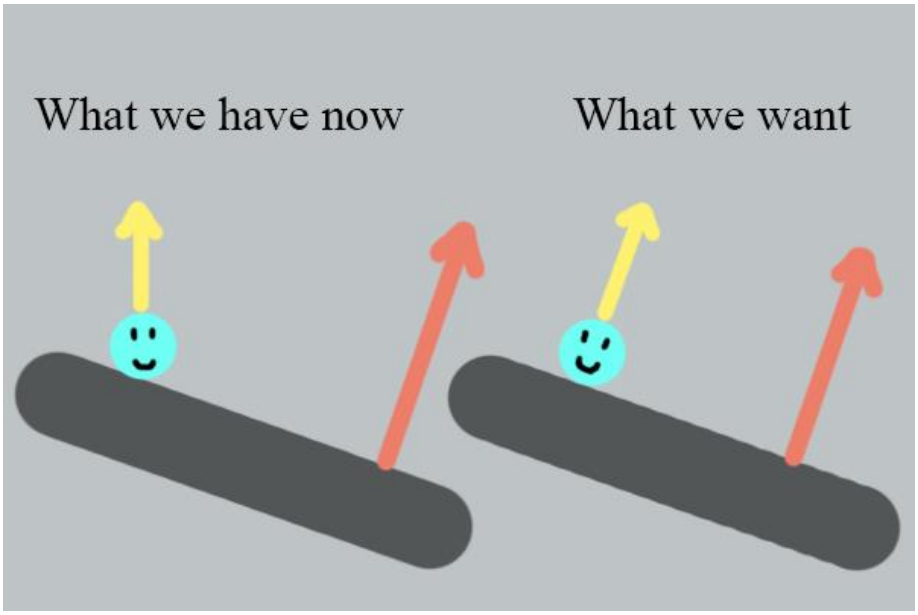

Fig. 11 - Wanted items inclination

We will perform this at the same time that the penguin Y position is adjusted, with the following code:

```
void FixedUpdate () {
    ...
    // Here up the rotation of the platform was set
    ...
    ...
    for(int i=0; i < onPlatformTransforms.Length; i++){
        if (!onPlatformIsFalling [i]) {
            ...
            if (hit.collider != null) {
                // Set the right inclination depending on the
platform
                float currentItemRotation = onPlatformTransforms
[i].eulerAngles.y;
                onPlatformTransforms [i].eulerAngles = new
Vector3 (-platform.eulerAngles.x, 180, -platform.eulerAngles.z);
                onPlatformTransforms [i].Rotate (new
Vector3(0,currentItemRotation-180,0));

                // Put the penguin at that position
                ...
            }
            ...
        }
    }
}
```

The process seems confusing but it is really simple: Firstly, we save the Y component of the penguin rotation, as we want to keep it and change the others two.

Secondly, we set the X and Z components of the rotation of the penguin depending on the platform rotation, and set 180 as Y value, because 180 is the original Y value of its rotation, the one that put it facing to the camera. Now we have the penguin with the right inclination, the same that the platform has. Lastly, we rotate it around its own Y axis to recover the facing direction that it had before this process.

We manage X and Z components of the rotation with the code above, so now we will manage Y component. We want penguins to realistically turn to face in the same direction they are moving to. We have done it in the script that each penguin owns. The code is the following:

```
public float speed = 50;
private float turnSpeed;

void Awake () {
      turnSpeed = 1.5f + Mathf.Abs(speed)/50; // This values give
a nice turn speed depending on the movement speed of the penguin

}

void FixedUpdate () {
    // Here up was implement the movement of the penguin
    ...

    // Here starts code to turn the penguin
    Vector3 myForward = Vector3.Normalize (new Vector3
(myTransform.forward.x, 0, myTransform.forward.z)); // In a
horizontal plane
    Vector3 myTargetForward = Vector3.Normalize (new Vector3
(Gyroscope.myOwnGravity.x, 0, Gyroscope.myOwnGravity.z)); // In
a horizontal plane

    // For the Green Penguin
    if (speed < 0)
        myTargetForward *= -1;

    // We use Cross Multiplication for lower speed when current
forward and target forward are close to be parallel, which looks
more realistic
    float direction = Vector3.Cross (myForward,
myTargetForward).y;
    float rotation = turnSpeed * direction; // "direction" also
increments or reduce the turn speed

    // Turn it
    myTransform.eulerAngles = new Vector3
(myTransform.eulerAngles.x, myTransform.eulerAngles.y +
rotation, myTransform.eulerAngles.z);

    // If it turned too much, come back
    myForward = Vector3.Normalize (new Vector3
(myTransform.forward.x, 0, myTransform.forward.z));
    float direction2 = Vector3.Cross (myForward,
myTargetForward).y;
    if ((direction > 0 && direction2 < 0) || (direction < 0 &&
direction2 > 0))
```

```
        myTransform.eulerAngles = new Vector3
(myTransform.eulerAngles.x, myTransform.eulerAngles.y -
rotation, myTransform.eulerAngles.z);
}
```

Code above is fully commented. To get a realistic turn, it has not been made constant or with lerp [15], but with a Vector3 cross multiplication. This means, penguin turns faster if is positioned perpendicular to its movement, and slower if it is positioned parallel to its movement, no matter positively or negatively. Result looks very natural and realistic.

Now that we have covered penguins' behavior over platform, let's cover what happens when they get away of it. As seen before, when this happens, a function called "SetFalling()" will be called into the individual penguin script.

```
private bool isFalling = false;

private bool isInWater = false;

private float yWater;

private Vector3 fallAndSwimDirection;

void Awake(){
    yWater = GameObject.FindGameObjectWithTag
("Water").GetComponent<Transform> ().position.y;
    yWater--; // We subtract 1 for noticing water once penguins
are INSIDE it
}


public void SetFalling(){
    isFalling = true;
    myRigidbody.constraints = RigidbodyConstraints.None; //
Rotation was constrained because we were managing it manually
    fallAndSwimDirection = Vector3.Normalize(new
Vector3(myRigidbody.velocity.x, 0, myRigidbody.velocity.z));
}

public void InWater(){
    isInWater = true;
}


void FixedUpdate () {

    if (!isInWater) {

        // If we are over the platform
        if (!isFalling) {

            // The code seen before
            ...

        }

        // If we are falling to the water
```

```
        else {
            if (myTransform.position.y < yWater)
                InWater ();
            else {
                // This line fix a bug at which sometimes
penguins kept static over the border of the platform
                if ( !(myRigidbody.velocity.y < -1))
myRigidbody.velocity += fallAndSwimDirection/4;

                // Fall
                myRigidbody.AddForce (new Vector3 (0,
Gyroscope.myOwnGravity.y, 0)); // Now, we have set
Gyroscope.myOwnGravity.y to -500 in the main script
            }
        }
    }

    // If we are already into the water
    else {
        // TODO
    }
}
```

As it can be read, we simply apply a constant force down when falling. We save the direction at which penguin started falling for later using that direction as swimming direction. That value has been also used to apply a smooth force to the penguins that may get blocked at the border of the platform, which may look as a bug, now fixed.

Once penguin is into water, let's see how we make it swim away:

```
...

private Vector3 swimStep;

public void SetFalling(){
    ...
    swimStep = fallAndSwimDirection / 12;
}

public void InWater(){
    ...
    myAnimator.SetTrigger ("Swimming");
    // We freeze all to get full control over the swimming
behaviour
    myRigidbody.constraints = RigidbodyConstraints.FreezeAll;
    myRigidbody.velocity = new Vector3 (0,0,0);
}


void FixedUpdate () {
    if (!isInWater) {...}

    // If we are already into the water
    else {
        // Manage movement and rotation
        myTransform.position += swimStep;
```

```
            myTransform.forward = Vector3.RotateTowards
(myTransform.forward, fallAndSwimDirection, 0.03f, 1);
        }
}
```

We change the penguin current animation to the "swimming" one, and move its position towards "fallAndSwimDirection". In order to manage its rotation, we use "Vector3.RotateTowards()" and constantly change the penguin initial rotation for facing towards the direction it is swimming.

Last but not least, we add some particle systems to make it more visually attractive. First, we add a snow dust particle system do emit when penguins are displacing over the platform. It has been used the default particle system of Unity [18].

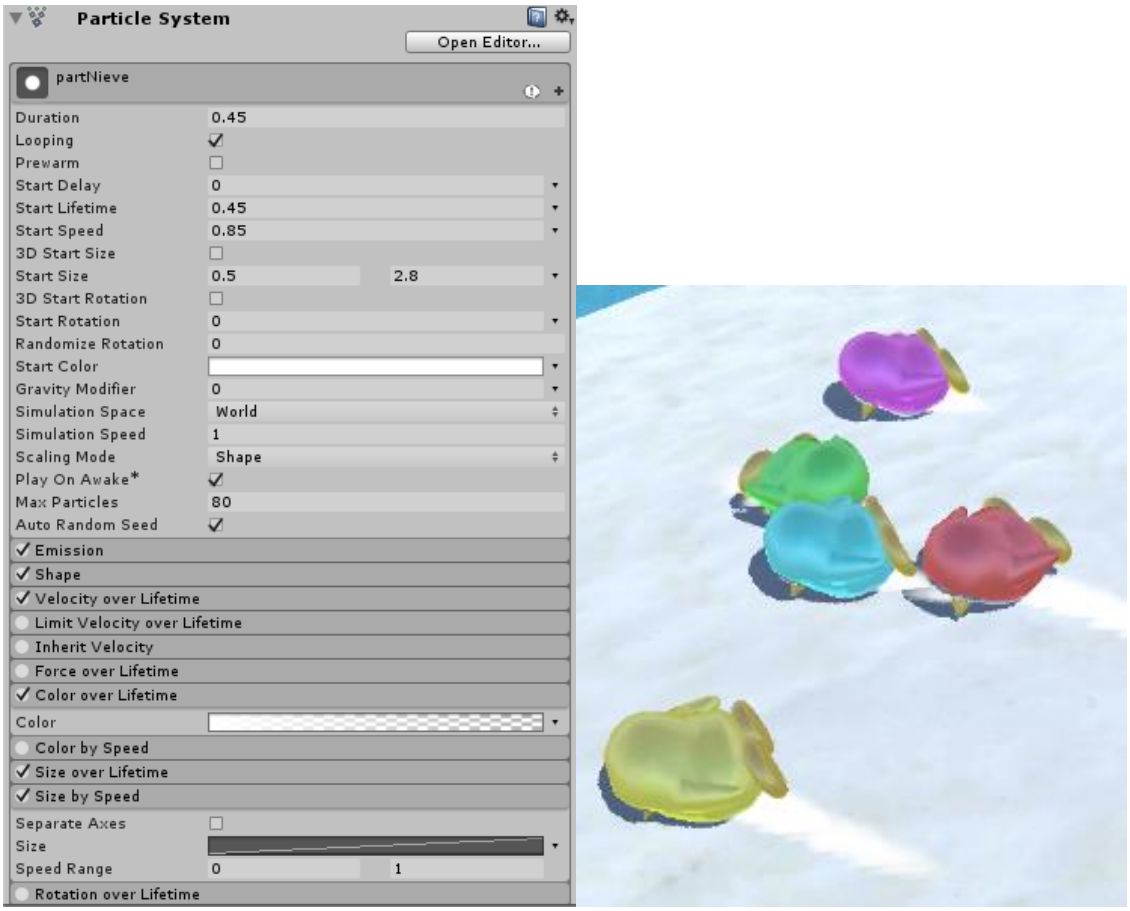

Fig. 12 - Snow dust particle system

As an end, we add a water splash particle system to the moment at which penguins get in touch with the water. It has been used the Ellipsoid Particle Emitter [19], as it is usually used for water particle systems due to its spherical nature.

Emit	☐
Min Size	0.3
Max Size	1
Min Energy	0.7
Max Energy	0.7
Min Emission	300
Max Emission	500
World Velocity	X 0 Y 0 Z 0
Local Velocity	X 0 Y 12 Z 0
Rnd Velocity	X 0 Y 0 Z 0
Emitter Velocity Scale	0.05
Tangent Velocity	X 0.18 Y 10 Z 0.06
Angular Velocity	0
Rnd Angular Velocity	6.36
Rnd Rotation	☑
Simulate in Worldspac	☑
One Shot	☐
Ellipsoid	X 0.01 Y 0.01 Z 0.01
Min Emitter Range	0

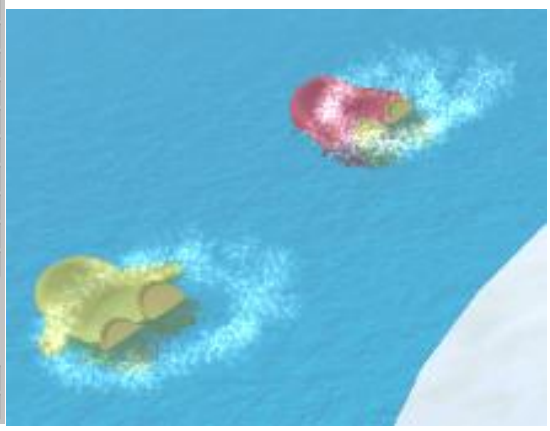

Fig. 13 - Water Splash Ellipsoid Particle System

4.5 In-level Elements

In this chapter, we are going to implement the in-level elements described below in the Game Design Document. We already have our usual penguins working, but the game needs more elements for entertaining.

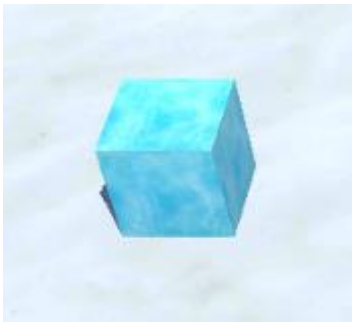

Fig. 14 – Ice Cube

Cubes must be simple elements which do not move at all when platform rotates, but which can be pushed by penguins. When they fall to the water, they must not "swim away" as penguins would, but slowly sink down. They will own a simplified version of the script that penguins have:

```
...
void FixedUpdate () {
    if (!isInWater) {
        // Over platform
        if (!isFalling) {
            // Nothing happens
        }
        // Falling
        else {
            if (myTransform.position.y < yWater)
                InWater ();
            else {
                // Fix the bug of keeping over the border
                if (!(myRigidbody.velocity.y < -1))
myRigidbody.velocity += fallDirection/4;

                // Fall
                myRigidbody.AddForce (new Vector3 (0,
Gyroscope.myOwnGravity.y, 0));

                // Emit splash particle from yWater+1.5 to
yWater
                if (myTransform.position.y < yWater + 1.5f) {
                    myWaterSplashTransform.eulerAngles = new
Vector3 ();
                    myWaterSplash.Emit (25);
                }
            }
        }
    }
```

```
    // Into water
    else {
        // Slowly sink down
        myRigidbody.AddForce (new Vector3 (0,
Gyroscope.myOwnGravity.y / 6, 0));
    }
}
public void InWater(){...}
public void SetFalling(){...}
```

This simpler script will implement the movement behavior of every static element over platform, while dynamic elements (penguins) will use the one seen in the chapter before. This one removes any line and variable related with movement or swimming, and makes elements slowly sink down when they get into water.

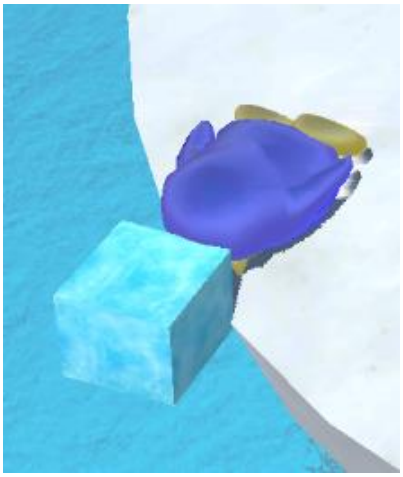

Fig. 15 - Ice Cube beign pushed away

Everything looks right, but does not work right. When rotating the platform, cubes lightly change their position over the platform depending on its inclination, as they actually work in a horizontal plane and are projected over an inclined platform. It would be unnoticeable if the platform wouldn't have a texture, but if does, and this fact does not affect gameplay but looks really ugly. This may also happen with penguins, but it is totally unnoticeable because they are in movement.

In order to fix this weird behavior, let's improve the script that ray casts and projects elements over the platform:

```
void FixedUpdate () {

    // Management of input and rotation of the platform
    // ...

    // Save upper central point of the platform for using later
```

```
    Vector3 platformTop = platform.position + platform.up *
platform.lossyScale.y;

    // Elements placement over the platform
    for(int i=0; i < onPlatformTransforms.Length; i++){
        if (!onPlatformIsFalling [i]) {

            // Only for static elements: Save relative position
before projecting
            float distanceToCenter = 0;
            float degreesToForward = 0;
            Vector3 pivot = platform.up;
            if (!onPlatformIsPingu [i]{

                // Obtain distanceToCenter, value to fix
distance from the element to the center of the platform
                distanceToCenter = Vector3.Magnitude
(onPlatformTransforms [i].position - platform.position);

                // Obtain degreesToForward (from the vector that
joins the element and the center of the platform, to the forward
vector of the platform)
                // Value to fix rotation of the element position
around Y axis of the platform
                platform.eulerAngles =
rotationJustBeforeCurrentRotation;
                Vector3 oldPlataformaTop = platform.position +
platform.up * platform.lossyScale.y;
                degreesToForward = Vector3.Angle
(onPlatformTransforms [i].position - oldPlataformaTop,
platform.forward);
                platform.eulerAngles = currentRotation;

                // For obtaining same result for elements at the
right half of the platform
                if (Vector3.Cross (onPlatformTransforms
[i].position - oldPlataformaTop, platform.forward).y < 0)
                    pivot *= -1;
            }

            // Perform raycast
            // ...

            if (hit.collider != null) {

                // Project position and adjust inclination
                // ...

                // Only for static elements: Set the relative
position that we had saved before projecting
                if (!onPlatformIsPingu [i]) {
                    // Fix Position
                    Vector3 fromCenter = onPlatformTransforms
[i].position - platform.position;
                    onPlatformTransforms [i].position =
platform.position + (fromCenter / Vector3.Magnitude
(fromCenter)) * distanceToCenter;
                    // Fix Circular Position
                    float newDegreesToForward = Vector3.Angle
(onPlatformTransforms [i].position - platformTop,
platform.forward);
```

```
                    onPlatformTransforms [i].RotateAround
(platformTop, pivot, newDegreesToForward - degreesToForward);
                }
            }
            else {
                // Set falling
                // ...
            }
        }
    }
}
```

This complex code really achieves what intended: to keep exactly the same position of the static elements (such as cubes) relative to the platform surface when the platform rotates. For doing so, we save two values before projecting a static element over the platform, and after being projected we adjust its position to maintain those two values equal as before. These values are the distance from the static element to the center of the platform; and the angle between the vector that joins the static element with the top center point of the platform, and the forward vector of the platform (it could also have been the right vector of the platform, it would not mind as long as that vector were perpendicular to the up vector of the platform). With the operations coded before, we keep those two values equal after projecting, and the static element keeps immutable as long as it is not being pushed by any penguin. With this, we have finished programing the behavior of our cube, as such as the static behavior of following elements, which will be kind of cubes with more functionalities.

Fig. 16 - Long Ice Cube

Long Ice Cubes are cubes than should only be able to move when pushed by a minimum determined number of penguins. This behavior can be implemented by triggering a collider a bit bigger than the Long Ice Cube, coded as follows:

```
public int pingusNeeded;
private int count = 0;
private Rigidbody myRigidbody;
private Cube myScript;

void Start () {
    myRigidbody = GetComponent<Rigidbody> ();
```

```
    myScript = GetComponent<Cube> ();
}

void OnTriggerEnter(Collider other){
    if ( (!myScript.GetIsFalling()) &&
other.GetComponents<Pingu> ().Length > 0) {
        count++;
        if (count >= pingusNeeded)
            myRigidbody.constraints =
RigidbodyConstraints.FreezeRotation;
    }
}

void OnTriggerExit(Collider other){
    if ((!myScript.GetIsFalling()) && other.GetComponents<Pingu>
().Length > 0) {
        count--;
        if (!(count >= pingusNeeded) &&
!myScript.GetIsFalling())
            myRigidbody.constraints =
RigidbodyConstraints.FreezeAll;
    }
}
```

That simple code freezes the rigidbody constraints of the Long Ice Cube when it is not being pushed by the required number of penguins. We will add more characteristics to this code later, as Bombs should be able to move it with no help if they explode next to it.

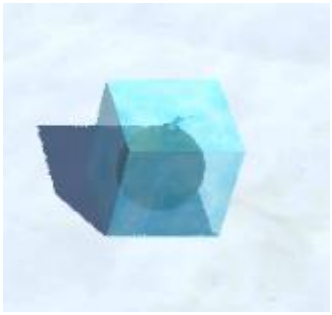

Fig. 17 - Bomb Cube

Bomb Cubes must explode when touched by any other element. When exploded, the causing penguins must receive a strong force as a reaction that might easily push it out of the platform. When exploded, Bomb Cubes must release a particle simple with sparks.

Fig. 18 - Bomb Cube exploded

Next is the implementation:

```
private Transform myTransform;
private float pushStrength = 70000;

void Awake () {
    myTransform = GetComponent<Transform> ();
}

void OnTriggerEnter(Collider other){
    // If it is a game element
    if (other.GetComponent<Penguin> () != null ||
other.GetComponentInParent<Cube> () != null) {

        // Calculate the direction towards which the element
must be pushed
        Vector3 otherPos = other.GetComponent<Transform>
().position;
        Vector3 dirPush = otherPos - myTransform.position;
        dirPush = Vector3.Normalize (new Vector3(dirPush.x, 0,
dirPush.z));

        // If it is a Long Ice Cube, push it without any extra
help
        LongIceCube bigCubeTrigger =
other.GetComponentInParent<LongIceCube> ();
        if (bigCubeTrigger != null) {
            bigCubeTrigger.Pushed ();
        }

        // Perform the push force
        other.GetComponent<Rigidbody> ().AddForce (dirPush *
pushStrength);

        Explode();
    }
}

void Explode() {
    // Do not move this unused transform form now on
    GetComponent<Rigidbody> ().constraints =
RigidbodyConstraints.FreezeAll;
```

```
    // Play spark particles
    ParticleSystem explosionParticles =
GetComponentInChildren<ParticleSystem>();
    explosionParticles.Play();

    // Make bomb invisible, as it has already exploded
    MeshRenderer mesh = GetComponentInChildren<MeshRenderer>();
    mesh.gameObject.SetActive(false);
}
```

Code has been meticulously commented to explain every single operation. Now, let's improve the script of the Long Ice Cube to let it be pushed by a single bomb without extra help, as mentioned before:

```
private bool pushedByBomb = false;
private float timeWhenPushedByBomb;
private int timeToWaitAfterPushedByBomb = 1; // Time to wait
before checking if I should be stopped
...

void OnTriggerEnter(Collider other){...}
void OnTriggerExit(Collider other){...}

public void Pushed(){
    myRigidbody.constraints =
RigidbodyConstraints.FreezeRotation; // Unfreeze position
    pushedByBomb = true;
    timeWhenPushedByBomb = Time.time;
}

void FixedUpdate(){

    // If I have been pushed by a bomb more than
timeToWaitAfterPushedByBomb second ago...
    if (pushedByBomb && Time.time - timeWhenPushedByBomb >=
timeToWaitAfterPushedByBomb){

        // If I am still moving but I am not being pushed by
penguins, gradually decrease my speed
        if (myRigidbody.velocity.magnitude > 0.1f && !(count >=
pingusNeeded) && !myScript.GetIsFalling()) {
            myRigidbody.velocity *= 0.8f;
        }

        // If I stopped moving, stop checking
        else if (pushedByBomb && !myScript.GetIsFalling()) {
            pushedByBomb = false;

            // If I am not being pushed by penguins, freeze
position
            if (!(count >= pingusNeeded*3))
myRigidbody.constraints = RigidbodyConstraints.FreezeAll;
        }
    }
}
```

The code bellow basically unfreezes position displacement of the Long Ice Cube for properly receive the push force of the bomb, and checks also when should freeze it again, reducing its speed if it is not being pushed by penguins. Both Bomb Cube and Long Ice Cube are completely implemented now.

Fig. 19 - Black Penguin

Black Penguins inflate when they are touched by other penguin, and later deflate to recover their original position. They inflate so fast that generate a push force to every element that they touch while inflating, a push force similar to a bomb.

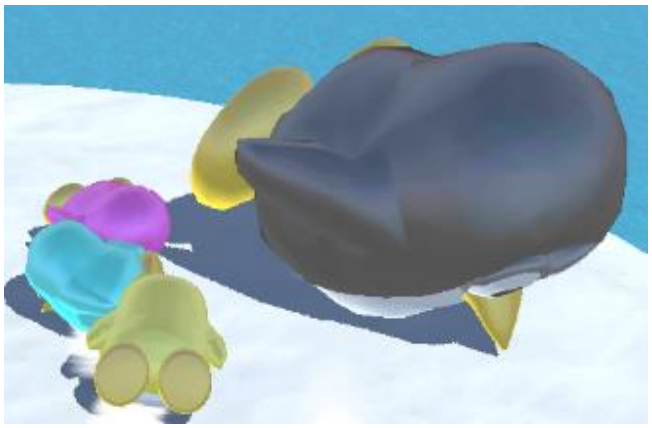

Fig. 20 - Black Penguin inflating

Implementation is a huge extension of the bomb implementation:

```
private float pushStrength = 35000; // Less than Bomb Cubes

private enum State {NORMAL, INFLATING, DEFLATING};
private State state;

private Vector3 initialScale;
private Vector3 maxScale;
```

```
private float velInflate = 20;
private float velDeflate = 5;

// These two will be "new Vector3(1, 1, 1) * vel"
private Vector3 vectorInflate;
private Vector3 vectorDeflate;

// I should not push an element more than one when inflating
// I save which elements did I already push since I started
inflating
private int numPushingTargets;
private int indexPushed = 0;
private GameObject[] pushed;

// I could be already in contact with non-penguins elements when
a penguin touches me
// All these elements must be pushed when I start inflating
private ArrayList cubesInContact;

...

void Start () {
    // Initialize everything
    // ...
}

void Update () {
    switch (state){

    case State.NORMAL:
        break;

    case State.INFLATING:
        myTransform.localScale += vectorInflate *
Time.deltaTime;

        // Start deflating and clean "pushed" array
        if (myTransform.localScale.x >= maxScale.x) {
            state = State.DEFLATING;
            pushed = new GameObject[numPushingTargets];
            indexPushed = 0;
        }
        break;

    case State.DEFLATING:
        myTransform.localScale += vectorDeflate *
Time.deltaTime;

        // Finish deflating
        if (myTransform.localScale.x <= initialScale.x){
            state = State.NORMAL;
            myTransform.localScale = initialScale;
        }
        break;
    }
}

void OnTriggerExit(Collider other){

    // Manage cubesInContact array
```

```
    if (state != State.INFLATING &&
other.GetComponentInParent<Cube> () != null &&
cubesInContact.Contains(other.gameObject)) {
        cubesInContact.Remove (other.gameObject);
    }
}

void OnTriggerEnter(Collider other){

    // Manage cubesInContact array
    if (state != State.INFLATING &&
other.GetComponentInParent<Cube> () != null &&
!cubesInContact.Contains(other.gameObject)) {
        cubesInContact.Add (other.gameObject);
    }

    // If a penguin touches me, start inflating
    if (state == State.NORMAL && other.GetComponent<Penguin> ()
!= null && ! myScript.GetIsFalling()) {
        state = State.INFLATING;

        //push every cube in contact
        foreach (GameObject cube in cubesInContact) {
            PushWithExplosionTo (cube);
        }
        cubesInContact = new ArrayList ();
    }

    // If inflating, push this penguin, and every following
element while inflating
    if (state == State.INFLATING
        && (other.GetComponent<Penguin> () != null ||
other.GetComponentInParent<Cubito> () != null)
        && ! pushed.Contains(other.gameObject)) {

        PushWithExplosionTo (other.gameObject);

        // Do not push this element again while this inflating
        pushed [indexPushed] = victim.gameObject;
        indexPushed++;
    }
}

private void PushWithExplosionTo(GameObject victim){
    // Calculate direction and push, like Bomb Cubes
    // ...

    // Do not push this element again while this inflating
    pushed [indexPushed] = victim.gameObject;
    indexPushed++;

}
```

It seems long but it is easy to understand by reading comments. The script checks which non-penguin elements are touching it, and when a penguin touches it, that penguin and all those elements receive the force. While inflating, new elements that may touch it also receive the force. The script remembers which elements has already pushed since started inflating for not pushing them again.

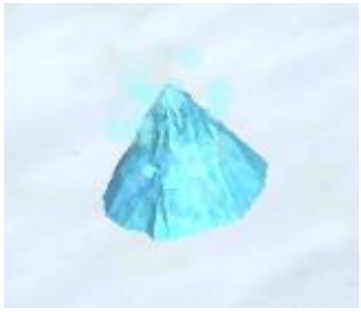

Fig. 21 - Tiny Extra Cold Iceberg

Tiny Extra Cold Icebergs are static elements that explode when are touched by a penguin, resulting on freezing that penguin for few seconds. When frozen, the penguin should become a static element that can not by moved neither by the platform rotation nor the push or any element.

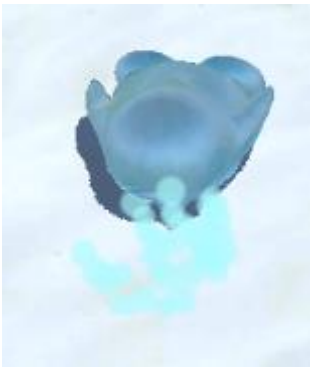

Fig. 22 - Tiny Extra Cold Icebergs exploding and freezing a penguin

Next is the implementation:

```
public Material materialFreeze;

private float timeWhenIWasFrozen = -1; // -1 means I was not
frozen yet
private float secsToUnfreeze = 2.5f;

private Rigidbody pinguRigidbody;
private Animator pinguAnimator;
private SkinnedMeshRenderer[] meshes;
private Material[][] savedMaterials;

void OnTriggerEnter(Collider other){
```

```
    if (other.GetComponent<Pingu> () != null) {

        // Freeze

        meshes =
other.GetComponentsInChildren<SkinnedMeshRenderer> ();
        savedMaterials = new Material[meshes.Length][];


        for (int i = 0; i < meshes.Length; i++){

            // We save every material of every mesh of the
penguin
            Material[] materials = meshes[i].materials;
            savedMaterials [i] = meshes[i].materials;

            // We set every material of every mesh of the
penguin to frozen
            for (int j = 0; j < materials.Length; j++) {
                materials[j] = materialFreeze;
            }

            meshes[i].materials = materials;

        }

        timeWhenIWasFrozen = Time.time;

        // Freeze rigidbody
        pinguRigidbody = other.GetComponent<Rigidbody> ();
        pinguRigidbody.constraints =
RigidbodyConstraints.FreezeAll;

        // Freeze animations
        pinguAnimator = other.GetComponent<Animator> ();
        pinguAnimator.enabled = false;

        // Freeze tiny iceberg rigidbody
        GetComponent<Rigidbody> ().constraints =
RigidbodyConstraints.FreezeAll;

        Explode();
    }
}

void Explode() {
    // Show explosion particles
    ParticleSystem[] exps =
GetComponentsInChildren<ParticleSystem>();
    foreach(ParticleSystem exp in exps) exp.Play();

    // Make me disappear
    MeshRenderer mesh = GetComponentInChildren<MeshRenderer>();
    mesh.gameObject.SetActive(false);
}

void Update(){
    if (timeWhenIWasFrozen != -1) {
        float currentFrozenTime = Time.time -
timeWhenIWasFrozen;
        if (currentFrozenTime >= secsToUnfreeze) {
```

```
                // Unfreeze

                // Recover saved materials
                for (int i = 0; i < meshes.Length; i++)
                    meshes [i].materials = savedMaterials [i];

                // Unfreeze all
                pinguRigidbody.constraints =
RigidbodyConstraints.FreezeRotation;
                pinguAnimator.enabled = true;

                // I have finished my work
                this.enabled = false;
            }
        }
}
```

As it can be read, when a penguin collision is triggered, this script store its materials and change them by a frozen material. Then, it freezes the rigidbody and animations of the penguin, and then explodes. After the defined seconds, it recovers every value to normal and disables itself.

All planned level elements have been correctly developed. Few more could be developed at future work, but these are enough to make an entertaining game.

4.6 Game Management

Game logics must have few things into accounts:

- Did the player penguin fall away? Then, player loses.
- Did everything but the player penguin fall way? Then, wait few seconds (player penguin might fall away due to inertia), and if player penguin is still over the platform, then player wins.
- Did game ended? Tell the player.
- Did player win? Then show him how much time he or she spent, and whether it is a new record or not.

In order to avoid frame rate counting how many elements are left, best is to make a call to the game management script from a game element when it falls away or simply disappears (such as bombs when exploding). We need to add that that call to every element script, and code our game management script:

```
// Global variables and their initialization
// ...

void Update(){
    // Tell player how many elements are left
    numElementsText.text = numElements.ToString ();

    // Did player win and he or she survived "victoryTimeDelay"
seconds?
    if (timeOfTheVictory != -1 && Time.time - timeOfTheVictory >
victoryTimeDelay && !playerIsFailling) {
        Victory ();
    }

    // Did player fall?
    else if (platerDidFall) {
        Fail ();
    }
}

// Called from the script that projects elements over the
platforms, as it did already count them
public void SetNumElements(int num){
    numElements = num - 1; // Player penguin is not counted
}

// Called from every element when falls or disappears
public void OneLessElement(){
    numElements--;
    if (numElements == 0)
        timeOfTheVictory = Time.time;
}

// Called from the penguin script that know it is the player
public void PlayerIsFalling(){
    playerIsFailling = true;
}
```

```
public void PlayerInWater(){
    platerDidFall = true;
}


private void Fail(){
    // Tell the player
    // ...
    FinalInCommon ();
}

private void Victory(){
    // Spent time, and check if it is a record
    int spentTime = Mathf.CeilToInt(timeOfTheVictory -
timeOfTheStart);
    bool newRecord = SavedData.IsRecord (spentTime);

    // Tell the player
    // ...

    FinalInCommon ();
}

private void FinalInCommon(){
    // Activate buttons to play again or return
    // ...

    // My work has ended
    this.enabled = false;
}

public void StartCountingTime(){
    timeOfTheStart = Time.time;
}
```

Code is meticulously. It checks if time spent is a new record from a saved data file, which is not coded yet. It will be coded some at its corresponding chapter: “Saving Encrypted Data”. For the rest, game management is finished.

4.7 Fixing Android Audio Delay

It is known that the Unity3D audio player system, although working right at the pc editor, does not works properly on android built apps, as there is a noticeable latency from the moment that the audio clip is played to the moment the audio clip actually starts sounding. This issue has not officially recognized by Unity3D, although they may probably be working on it for future versions of the game engine. However, it has been widely noticed by Unity3D users [6].

After some researching, a solution has been encountered: do not use the Unity3D audio player system but the Android Sound Pool [20] for managing and using sound files. In search of performance, only those audio files that need zero latency must be loaded to the Android Sound Pool to save memory in use. Those whose latency does not matter are best loaded using the Unity3D audio player system, as it makes a good loading and releasing use of the memory. For an easier way to manage native android functions, it was used following plugin: Android Audio Bypass [21], which have the Apache 2 license [22].

In order to load files by using the Android Sound Pool, they must not be compressed into Unity3D default format. Unity compresses every audio file to that format, so we need to avoid it. We do so by creating a Streaming Assets folder [23] and importing our audio files to that folder. That way, Unity3D will not comprise them.

As we are playing the audio without the aid of the Unity3D audio source component help, we need to manually audio modifiers as public parameters of a script, as well as the name of the audio file to be found into the Streaming Assets folder.

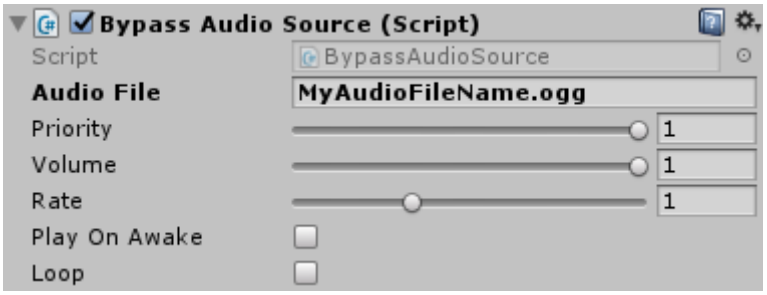

Fig. 23 - Script that manages some of the audio parameters an Audio Source component would

As a last step to get ready, for a correct use of the plugin, a unique call must be done at the very beginning to register the sound file: “audioBypassInstance.Call<int>("register", soundFile)”. Finally, whenever we want to play the audio file, a play call must be done to the plugin: “audioBypassInstance.Call<int>("play", soundId, leftVolume, rightVolume, priority, loop, rate)”. The plugin will next natively use the Android Audio Pool to play the sounds.

4.8 Saving Encrypted Data

The vast majority of video games locally save player's data as achievement, user data, continue point, records, and so on, to avoid starting from zero every time the game is played. This is a priceless functionality for users, as they may other way feel they are losing their time when playing. However, this is also a dangerous tool: players may find the way to manually change their saved data to cheat their progress. Maybe developers should not try to avoid this cheating issue, as the main objective is players having fun, and they may have funnier by cheating their games. However, this game tries to have players sanely comparing their results with friends who also play the game, and with cheats this wouldn't be possible, as people who do not cheat would feel frustrated and could stop playing, and people who cheat may feel they have obtained everything with no effort and it is time to change to another game. In conclusion, we want to avoid players cheating out game, so we will save game data with an extremely safe way: an external encrypted file.

First, let's see how can we write our external file to use it to store game data:

```
private static void writeStringToFile( string str, string
filename )
{
    // Access to the file
    string path = pathForDocumentsFile( filename );
    FileStream file = new FileStream (path, FileMode.Create,
FileAccess.Write);

    // Write into the file
    StreamWriter sw = new StreamWriter( file );
    sw.WriteLine( str );

    // Close the file
    sw.Close();
    file.Close();
}


private static string readStringFromFile( string filename)
{
    string path = pathForDocumentsFile( filename );

    if (File.Exists(path))
    {
        // Access to the file
        FileStream file = new FileStream (path, FileMode.Open,
FileAccess.Read);

        // Read the file
        StreamReader sr = new StreamReader( file );
        string str = sr.ReadLine ();

        // Close the file
        sr.Close();
        file.Close();

        // Return string
        return str;
    }
```

```
        else
        {
            // Return nothing
            return "";
        }
    }


    private static string pathForDocumentsFile( string filename )
    {
        // If we are in Android
        if(Application.platform == RuntimePlatform.Android)
        {
            string path = Application.persistentDataPath;
            path = path.Substring(0, path.LastIndexOf( '/' ) );
            return Path.Combine (path, filename);
        }

        // else
        // others paths...
    }
```

As it can be read, the path for storing the saved data file is got from "Application.persistentDataPath" [24], which is a directory path where data expected to be kept between runs can be stored. The rest of the code is the usual way to read and write files.

Thanks to this code, we can keep a string saved between different plays. Now, we want that string to be encrypted. One way is simply changing it to bytes via "System.Text.Encoding.Unicode.GetBytes(string)", and then getting a string of different base form those bytes. This way, there is no key needed to decrypt the file, but it is encrypted anyways and players who get to access the saved data file would not perceive that it is a saved data file or would not know how to cheat it. More secure ways to encrypt will be developed at future work.

Another improvement for this saving system would be to offer players a way to easy save several single strings with a key to access each one, as Player Prefs [25] does. We must then serialize the string that we are going to save, with special characters as separators. We have implemented it this way:

```
    public static string LoadSingleData(string DataName){
        string data = readStringFromFile (filename);
        data = Decrypt (data);

        // We split the data into pairs key-data
        char[] charSeparator = new char[] {'#'};
        string[] pairsData = data.Split (charSeparator,
    StringSplitOptions.RemoveEmptyEntries);

        charSeparator = new char[] {'$'};
        foreach (string myPair in pairsData){
            // Separate key form data
            string[] CurrentPair = myPair.Split (charSeparator,
    StringSplitOptions.RemoveEmptyEntries);
```

```
        // We check every pair key-data to check if there is a
pair with the desided key, which is DataName
        if (CurrentPair [0] == DataName)
            return CurrentPair [1];
    }

    return null;
}

public static void SaveSingleData(string DataName, string
DataValue){
    string currentData = readStringFromFile (filename);
    currentData = Decrypt (currentData);

    string newData = "";

    // We split the data into pairs key-data
    char[] charSeparator = new char[] {'#'};
    string[] pairsData = currentData.Split (charSeparator,
StringSplitOptions.RemoveEmptyEntries);

    charSeparator = new char[] {'$'};
    foreach (string myPair in pairsData){
        // Separate key form data
        string[] currentPair = myPair.Split (charSeparator,
StringSplitOptions.RemoveEmptyEntries);
        // Here we rewrite everything from currentData but the
data that we are going to save, to avoid having it twice
        if (! (currentPair [0] == DataName))
            newData += "#" + myPair;
    }

    // And then write the new file with the righ serialization
    newData += "#" + DataName + "$" + DataValue;

    if (encrypt) newData = Encrypt (newData);

    writeStringToFile (newData, filename);
}
```

From now on, if we want to locally save single data, we can do it as easily as writing “SaveLoadData.SaveSingleData(key, value)”, and it will be safely scripted and stored.

Last but not least, let’s ease our saved data structure. We can have a clean structure storing most of our data into a serializable class, and then serialize the whole class into a single string using Json [26]. Next is the way to do it:

```
    [System.Serializable]
    public class ClassToStoreData{
        string data1;
        bool data2;
        int data3;
        ...
    }

    static ClassToStoreData _Data;
```

```
    static ClassToStoreData Data{
        get{
            if (_Data == null) // We make sure Data is loaded
when acceding it
                LoadDataJson ();
            return _Data;
        }
        set{
            _Data = value;
        }
    }

    string datakey = "myKey";

    public static void LoadOptionsJson(){
        _Data = new OptionsJson ();

        string json = SaveLoadData.LoadSingleData (datakey);
        if (json != null)
            JsonUtility.FromJsonOverwrite(json, _Data);
    }

    public static void SaveOptionsJson(){
        if (_Data == null) {
            return;
        }
        string json = JsonUtility.ToJson(_Data);
        SaveLoadData.SaveSingleData (datakey, json);
    }
```

Now we have our saved data not just secured but also cleanly structured. Results are even better than expected.

4.9 Csv Localization

We want our game localized to be playable with different languages and then reach the widest number of players possible. There are many ways to store translated texts, but one of the cleaner and simpler is by managing them such as spread sheet and then exporting it to the Csv format, which is a simple text file with fields separated by commas.

	A	B	C	D
1	key	English	Spanish	Japanese
2	Key1	Test 1	Prueba 1	Tesuto 1
3	Key2	Test 2	Prueba 2	Tesuto 2

Fig. 24 - Spread Sheet with translated texts for localisation

```
key,English,Spanish,Japanese
Key1,Test 1,Prueba 1,Tesuto 1
Key2,Test 2,Prueba 2,Tesuto 2
```

Fig. 25 – Exported Csv file read with a text editor

What we need know is a programming structure that can read and store the Csv rows into variables easy to access. There are several ways to do it as it can be read with a simple search on internet. Any of them can be used. The thing that is important to us is how to integrate it into Unity3D. First, we need to know which language the android system has:

```
private static string defaultLanguage = "English";
private static string _gameLanguage;

public static string gameLanguage{
    get{
         // We must initialize _gameLanguage first time we call
it
        if (_gameLanguage == null)
            Initialize ();
        return _gameLanguage;
    }
    set{
        _gameLanguage = value;
    }
}

private static void Initialize(){
    // We get language from the Android device
```

```
        gameLanguage = Application.systemLanguage.ToString();
    }
```

That is the way we get it. We must take into account how Unity3D names every language [27] to use those names for the first row of our Csv file. Now, we must access our Csv structure and get the desired text with the desired language, using the default language if the desired language is not available:

```
    public static string GetString(string key){
        string text = Localization.GetString(gameLanguage, key);

        // If gameLanguage is not into the Csv, set English as
    defaultLanguage
        if (text == key) {
            gameLanguage = defaultLanguage;
            text = Localization.GetString (gameLanguage, key);
        }

        // If gameLanguage is into the Csv, but this specific field
    is not translated, show it in English
        if (text == "") {
            text = Localization.GetString (defaultLanguage, key);
        }

        text = ReplaceSpecials (text);

        return text;
    }

    private static string ReplaceSpecials(string text){

        // Change of line
        text = text.Replace("\\n", "\n");

        return text;
    }
```

As it can be read, we need to replace those characters of the Csv that we want to have a special functionality or content. One essential is “\n” (change of line), as it is not recognized in that way when read from an external text file.

Finally, if want to easily change texts into the Unity3D Canvas, we can add them the following script as a component:

```
    private void Awake()
    {
        Text text = this.GetComponent<Text>();

        if (text != null)
            text.text = GetString (text.text);
    }
```

By using this, we only need to write as a text into de UI Text the key of our Csv localization file, and when the UI Text awakes, key will by changed for the corresponding text into the corresponding language.

4.10 Map of Levels Interactivity

This chapter will cover the implementation of the map of levels, which actually is as complex as the gameplay implementation is. This is due to the intention of making this part of the game especially attractive, where majority of games only have some buttons to access their levels, which are their only attractive component.

As explained in the Game Design Document, we want a 3D map of levels where player can scroll the camera by sliding a finger. We want the animated player penguin idling over one of the numerous tiny platforms representing the game levels. When touching a level, the penguin will go there by walking throw paths if it is close, of by swimming if it is far. We also want a different part of the map of levels working as a map of special levels, which are unlocked every 10 normal levels. Finally, we want a Stars scoring system, where player can get from 0 to 3 stars on every level depending on his or her spent time to win.

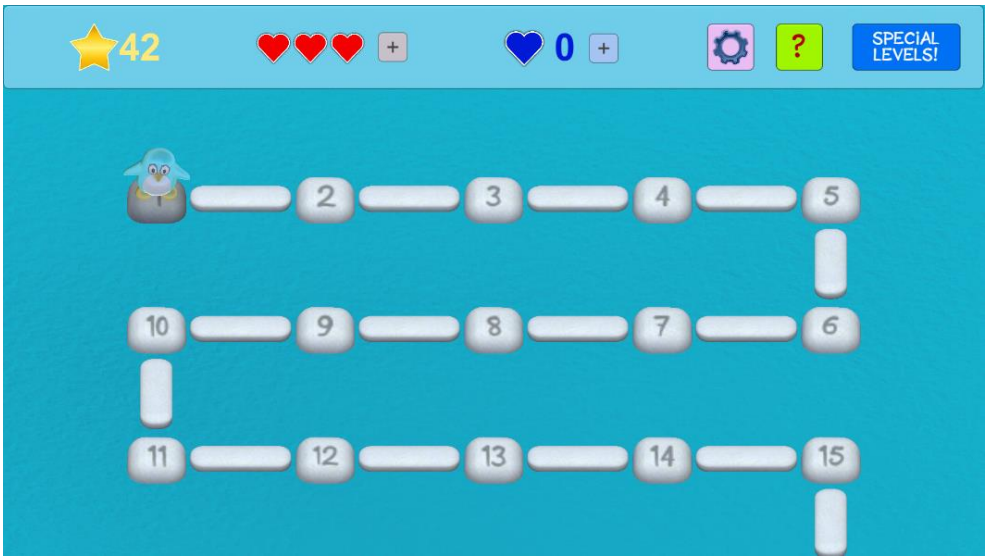

Fig. 26 - Map of Levels

We will see how to implement all of these characteristics on following sub-chapters.

4.10.1 Camera Movement

We want a forward scroll when sliding the finger over the screen. We also want that scroll continuing with inertia when releasing the finger, until finally stop. We need to ray cast our touch begin inputs to see if we are touching a level. We also need to ray cast our touch holding input to see if we are still over that level. We finally need to ray cast our touch end inputs to see if we release the finger over the same level to then consider that level as selected.

We manage all of that with the following implementation:

```
void Update () {

    // If camera is moving, we make sure that it does not go
beyond limits
    if (cameraRigidbody.velocity != new Vector3 ()) {
        cameraTransform.position = RestrictCameraZ
(cameraTransform.position);
    }

    if (pause)
        return;

    // If I touch the screen
    if (Input.touchCount > 0 && Input.GetTouch (0).phase ==
TouchPhase.Began) {

        // Stop camera
        cameraRigidbody.velocity = new Vector3 ();

        // If I had a level selected, deselect it
        if (focusObj != null) {
            nivelMesh.material.color = Color.white;
        }

        // Raycast to see if I am touching a level
        focusObj = null;
        Ray ray = Camera.main.ScreenPointToRay (Input.GetTouch
(0).position);
        RaycastHit hit;

        // If I select a level, show is as selected
        if (Physics.Raycast (ray, out hit, Mathf.Infinity)) {
            if (hit.collider.tag == "Level") {
                focusObj = hit.collider.gameObject;
                nivelMesh =
focusObj.GetComponentInChildren<MeshRenderer> ();
                nivelMesh.material.color = onTouchColor;
            }
        }

        // Else, move camera
        else {
            StartMovingCamera ();
        }

    }
```

```
    // If I hold touching after touching a level
    else if (focusObj && Input.touchCount > 0 && Input.GetTouch
(0).phase == TouchPhase.Moved) {
        Ray ray = Camera.main.ScreenPointToRay (Input.GetTouch
(0).position);
        RaycastHit hit;

        // Raycast to see if I am still touching that level
        if (Physics.Raycast (ray, out hit, Mathf.Infinity)) {
            if (hit.collider.gameObject == focusObj) {
                nivelMesh.material.color = onTouchColor;
            }
            // Else, deselect it, and move camera
            else {
                nivelMesh.material.color = Color.white;
                StartMovingCamera();
            }
        }

        // Else, deselect it, and move camera
        else {
            nivelMesh.material.color = Color.white;
            StartMovingCamera();
        }
    }

    // If I release the touch after touching a level
    else if (focusObj && Input.touchCount > 0 && Input.GetTouch
(0).phase == TouchPhase.Ended) {

        // Tell the player penguin it must go there
        pinguMap.SetDestiny(focusObj.GetComponent<Nivel>());

        // Deselect it as it already did what intended
        nivelMesh.material.color = Color.white;
        focusObj = null;
    }


    // If I hold touching after touching out of any level
    else if (moveCamera && Input.touchCount > 0 &&
Input.GetTouch (0).phase == TouchPhase.Moved) {

        // Check delta touch input position and the new position
of the camera
        Vector2 newTouchPosition = new
Vector2(0,Input.GetTouch(0).position.y);
        Vector3 auxCameraPosition = new
Vector3(cameraTransform.position.x, cameraTransform.position.y,
cameraTransform.position.z);
        auxCameraPosition += cameraTransform.TransformDirection
((Vector3)((oldTouchPosition - newTouchPosition) *
camera.orthographicSize / camera.pixelHeight * 2f));

        // If the new position is between the limits
        Vector3 auxCameraPositionFixed = RestrictCameraZ
(auxCameraPosition);
        if (auxCameraPosition == auxCameraPositionFixed) {

            // Set the new position
```

```
            cameraTransform.position = auxCameraPositionFixed;

            // Save the current touch position
            oldTouchPosition = newTouchPosition;

            // We make sure camera Y coordinate does not go far
away, although we would not notice it as it is orthographic
            while (cameraTransform.position.y < 24.5f)
                cameraTransform.position -=
cameraTransform.forward;
            while (cameraTransform.position.y > 25.5f)
                cameraTransform.position +=
cameraTransform.forward;


            // To obtain inertia, we save the speed of this last
delta position
            speedOnLastFrame = -Input.GetTouch
(0).deltaPosition.y / Input.GetTouch (0).deltaTime;
            speedOnLastFrame *= 0.03f;
        }
    }

    // If I release the finger after touching the water
    else if (moveCamera && Input.touchCount > 0 &&
Input.GetTouch (0).phase == TouchPhase.Ended) {

        // To obtain intertia
        cameraRigidbody.velocity = new Vector3 (0, 0,
speedOnLastFrame);

        // Stop manually moving the camera
        moveCamera = false;
    }
}

// Initialize movement values
private void StartMovingCamera(){
    focusObj = null;
    moveCamera = true;
    oldTouchPosition = new
Vector2(0,Input.GetTouch(0).position.y);
    speedOnLastFrame = 0;
}

// Stop camera if it go out of the limits
private Vector3 RestrictCameraZ(Vector3 camaraPosition){
    if (camaraPosition.z > initialZ) {
        cameraRigidbody.velocity = new Vector3 ();
        camaraPosition = new Vector3 (camaraPosition.x, 25,
initialZ);
    } else if (camaraPosition.z < finalZ) {
        cameraRigidbody.velocity = new Vector3 ();
        camaraPosition = new Vector3 (camaraPosition.x, 25,
finalZ);
    }
    return camaraPosition;
}
```

Code is long and bit complex but it is meticulously commented.

4.10.2 Levels and Special Levels Structure

We need to set a code structure to store every level with its corresponding index, because penguin will need to know which game object is the "next level" if it wants to walk forward, or which is the "previous level" if it wants to move backwards. Levels must also keep the information of their required times for player obtaining their corresponding stars. They finally need to know if they are normal or special, if they are unlocked or not, and where to place a "path" between it and the next level. We do all of that in the next way:

```
// Global lists that will contain every level
private static Dictionary<int,Nivel> LevelsList;
private static Dictionary<int,Nivel> SpecialLevelsList;

// myLevelsList will be NivelList or SpecialNivelList depending
on this level being being normal or special
private Dictionary<int,Nivel> myLevelsList{
    get{ return SpecialLevel ? SpecialLevelsList : LevelsList;}
    set{ if (SpecialLevel) SpecialLevelsList = value;
        else LevelsList = value;}
}

// Path model between levels
public GameObject Path;

public int LevelNum;

// Maximum time player at which player must complete a level to
obtain each star
public int Star1TimeGoal = 30;
public int Star2TimeGoal = 20;
public int Star3TimeGoal = 10;

//-1 if this level has not been player yet (from saved data)
private int recordTime = -1;

// True if this is a normal level. False if I this is a special
level
public bool SpecialLevel = false;

// True if player did already unlock this level
private bool unlocked;

private const int LEVELS_IN_A_ROW = 5;

void Awake(){

    // If I am a special level that is not visible yet due to
number of normal levels completed, do nothing
    if (SpecialLevel && LevelNum >
StaticData.OptionsData.numVisibleSpecialLevels) {
        this.gameObject.SetActive (false);
        return;
    }

    // Initialize static lists of levels
    if (myLevelsList == null) {
```

```
            LevelsList = new Dictionary<int,Nivel> ();
            SpecialLevelsList = new Dictionary<int,Nivel> ();
        }

        // If I have just initialize level lists, initialize also
the number of stars
        if (LevelsList.Count == 0 && SpecialLevelsList.Count == 0) {
            InterfaceMapValues.StarsCount = 0;
        }

        // Add this level to its corresponding list
        if (!myLevelsList.ContainsKey (LevelNum))
            myLevelsList.Add (LevelNum, this);

        // Get from saved data if this level is already unlocked
        int unlockedLevels = SpecialLevel ?
StaticData.OptionsData.numUnlockedSpecialLevels :
StaticData.OptionsData.numUnlockedLevels;
        unlocked = LevelNum <= unlockedLevels;

        // Get record from saved data
        if (LevelNum < unlockedLevels) {
            recordTime = SpecialLevel ?
StaticData.OptionsData.specialLevelsTimes [LevelNum - 1] :
StaticData.OptionsData.levelsTimes [LevelNum - 1];
        }

        // Physically show the number of this level over its
platform
        GetComponentInChildren<Text> ().text = "" + LevelNum;

        // If this level has been completed any time
        if (recordTime >= 0) {
            // Add its start to the global counter
            if (recordTime <= Star1TimeGoal) {
                InterfaceMapValues.StarsCount++;
                if (recordTime <= Star2TimeGoal) {
                    InterfaceMapValues.StarsCount++;
                    if (recordTime <= Star3TimeGoal) {
                        InterfaceMapValues.StarsCount++;
                    }
                }
            }
        }
    }

    void Start(){
        DrawPath ();
    }

    // Place a path model between thsi level and the next one
    private void DrawPath(){
        Level nextNivel = myLevelsList[LevelNum + 1];

        // If there is not a next level, or if the next level is
locked, do nothing
        if (nextNivel == this || !nextNivel.IsUnlocked())
            return;

        // Place the path
        Transform myPath = Instantiate (Path).transform;
```

```
        myPath.position = Middle (transform.position,
    nextNivel.transform.position);

        // Rotate it if it is at the border left or right
        if (LevelNum % LEVELS_IN_A_ROW == 0)
            myPath.Rotate (0, 90, 0);

        // Specials levels may change their position, so paths must
    move with them
        myPath.parent = this.transform.root;
    }

    private Vector3 Middle(Vector3 v1, Vector3 v2){
        return (v1 + v2) / 2;
    }
```

Now we have our levels nicely structured and we can access to them from any external script.

4.10.3 Character Movement

This may be one of the hardest implementations of the game. As mentioned before, when we touch a level, we want our penguin to go there by walking (if the level is 1 row close) or by diving (if the level is farther than 1 row close). If we touch a level, penguin must instantly transport to that point, to avoid impatient players waiting for the penguin movement. If that last case happens, or if player touches a level over which the penguin is already idling, the level info must be shown with the option to play it. We implement all of that as follows:

```
// Many global variables
//...

void Awake(){
    // Initialize global variables
    //...
}

void Start(){
    // Get current level from saved data
    currentLevel = Level.GetLevelFromList
(StaticData.OptionsData.lastLevelPlayedNum,
StaticData.OptionsData.lastLevelPlayedWasSpecial);

    // Some more initializations
    // ...

    // If we are in a especial level, place the camera over the
zone of special levels
    if (currentLevel.SpecialLevel) {
        myCamera.position = new Vector3(Level.GetLevelFromList
(1, true).transform.position.x + (Level.GetLevelFromList
(3).transform.position.x - Level.GetLevelFromList
(1).transform.position.x),myCamera.position.y,myCamera.position.
z);
    }

    // Place the camera to initially have our penguin into its
range
    int myRow = (currentLevel.GetLevelNum ()-1) /
LEVELS_IN_A_ROW; // First row is number 0
    myRow++; // First row is now number 1
    // Place camera to have the row of our penguin at its centre
    if (myRow > 2) {
        int rowsToMove = myRow - 2;
        int separationRows = GameObject.FindGameObjectWithTag
("Scripts").GetComponent<TouchOnMap> ().SeparationRows;
        myCamera.position -= new Vector3 (0,0, rowsToMove *
separationRows);
    }
}

// This is called from the script that manages touch inputs
// touchInput parameter will be false if this function has been
called internally
public void SetDestiny(Level level, bool touchInput = true){
```

```
    if (!level.IsUnlocked ())
        return;

    // If we select a level where the penguin already is, show
level info
    if (currentLevel == level && !onTheWay && divingState ==
DivingState.Nothing) {
        ShowLevelInfo (level);
        return;
    }

    MeshRenderer levelMesh =
level.GetComponentInChildren<MeshRenderer> ();

    // If we touch a level twice, transport the penguin
instantly to it and show level info
    if (currentLevel != level && levelMesh.material.GetFloat
("_Metallic") == metallicValueFinalLevelToGo && touchInput) {

        // Manage animation and snow particles
        myAnimator.SetTrigger ("ForceIdle");
        myAnimator.SetBool ("Walking", false);
        myAnimator.SetBool ("Jumping", false);

        // Manage particles
        mySnowParticles.Stop ();
        if (myWaterSplash.emit) {
            myWaterSplash.emit = false;

myWaterSplash.GetComponent<Transform>().localPosition -=
transform.up;
        }

        // Manage position
        myTransform.position = new Vector3
(level.transform.position.x, yPos, level.transform.position.z);
        myTransform.eulerAngles = new Vector3 (0, 180, 0);

        // Manage global variables
        currentLevel = level;
        myPreviousLevel = currentLevel.GetPreviousLevel ();
        myNextLevel = currentLevel.GetNextLevel ();
        onTheWay = false;

        // Manage diving state
        divingState = DivingState.Nothing;
        timer = 0;
        divingValuesSet = false;
        nextLevelToGoAfterDiving = null;

        //Finally, show level info
        ShowLevelInfo (level);

        return;
    }

    // Update the level at which penguin is going, and change
its material to notice it
    meshFinalLevelToGo.material.SetFloat("_Metallic", 0);
    meshFinalLevelToGo = levelMesh;
```

```
    meshFinalLevelToGo.material.SetFloat("_Metallic",
metallicValueFinalLevelToGo);

    // If penguin is currently diving, enqueue next level to go
    if (divingState != DivingState.Nothing) {
        nextLevelToGoAfterDiving = level;
        return;
    }

    finalLevelToGo = level.GetLevelNum ();

    // Go walking level by level
    if (isOneRowClose (level)) {
        if (!onTheWay) {
            onTheWay = true;
        }
        MoveToNextLevel ();
    }

    // Or going by diving
    else {
        nextLevelToGoDiving = level;
        divingState = DivingState.GoingIntoWater;
    }
}

private bool isOneRowClose(Level otherLevel){

    // If one level is normal and another is special, they will
be far with no doubts
    if (currentLevel.SpecialLevel != otherLevel.SpecialLevel)
        return false;

    int myRow = (currentLevel.GetLevelNum ()-1) /
LEVELS_IN_A_ROW;
    int theOtherRow = (otherLevel.GetLevelNum()-1) /
LEVELS_IN_A_ROW;
    if (Mathf.Abs (myRow - theOtherRow) <= 1)
        return true;

    return false;
}

private void MoveToNextLevel(){

    // Go forward
    if (finalLevelToGo > currentLevel.GetLevelNum ()) {
        nextLevelToGoWalking = myNextLevel;
        myPreviousLevel = currentLevel;
    }
    // Or go backward
    else if (finalLevelToGo < currentLevel.GetLevelNum ()) {
        nextLevelToGoWalking = myPreviousLevel;
        myNextLevel = currentLevel;
    }
    // Or stay a the same place
    else
        nextLevelToGoWalking = currentLevel;

    // Set some variables
```

```
    Transform level =
nextLevelToGoWalking.GetComponent<Transform> ();
    placeToGoNowWalking = new Vector3(level.position.x,
myTransform.position.y, level.position.z);

    ManageAnimation ();
}

private void ManageAnimation(){

    // Look at the fight direction of moving
    if (placeToGoNowWalking.x > myTransform.position.x) {
        myTransform.eulerAngles = new Vector3 (0, 90, 0);
    }
    else if (placeToGoNowWalking.x < myTransform.position.x) {
        myTransform.eulerAngles = new Vector3 (0, 270, 0);
    }
    else if (placeToGoNowWalking.z < myTransform.position.z) {
        myTransform.eulerAngles = new Vector3 (0, 180, 0);
    }
    else if (placeToGoNowWalking.z > myTransform.position.z) {
        myTransform.eulerAngles = new Vector3 (0, 0, 0);
    }

    // Animate and show snow particles
    myAnimator.SetBool ("Walking", true);
    mySnowParticles.Play ();
}

void ShowLevelInfo(Level level){
    //...
}

// Obtains level where penguin is or the one where it will be
just before start walking if it received the order
public Level GetCurrentLevelBeforeNextDiving(){
    Level result = currentLevel;
    if (onTheWay) {
        result = nextLevelToGoWalking;
    }
    else if (divingState != DivingState.Nothing) {
        result = nextLevelToGoDiving;
    }
    return result;
}

void Update(){

    // If penguin is walking
    if (onTheWay) {
        // Make the delta change of position
        float step = speed * Time.deltaTime;
        myTransform.position = Vector3.MoveTowards
(myTransform.position, placeToGoNowWalking, step);
        // If penguin reached its destiny
        if (myTransform.position == placeToGoNowWalking) {
            // Reset global variables
            currentLevel = nextLevelToGoWalking;
            myPreviousLevel = currentLevel.GetPreviousLevel ();
            myNextLevel = currentLevel.GetNextLevel ();
```

```
            // If penguin reached its destiny ir if it needs to
stop to star diving
            if (finalLevelToGo == currentLevel.GetLevelNum () ||
divingState != DivingState.Nothing) {
                // Stop everything
                onTheWay = false;
                myAnimator.SetBool ("Walking", false);
                myTransform.eulerAngles = new Vector3 (0, 180,
0);
                mySnowParticles.Stop ();
                return;
            }

            // Else, set the next level to continue walking
            MoveToNextLevel ();
        }
    }

    // If I am diving
    else if (divingState != DivingState.Nothing) {

        // Initialize diving values if they are not yet
        if (!divingValuesSet) {
            placeToGoNowDiving =
nextLevelToGoDiving.GetComponent<Transform> ().position;
            myAnimator.SetBool ("Jumping", true);
            Vector3 dir = placeToGoNowDiving -
myTransform.position;
            dir.Normalize ();
            myTransform.forward = new Vector3 (dir.x, 0, dir.z);
            dir *= JUMPING_DISTANCE;
            divingDir = new Vector3 (dir.x, -JUMPING_SPEED,
dir.z);
            divingValuesSet = true;
        }

        // Set spent time, to check when penguin gets totally
into water
        timer += Time.deltaTime;

        switch (divingState) {

        case DivingState.GoingIntoWater:

            // Jump down toward the water
            myTransform.position += divingDir * Time.deltaTime;

            // Splash particles
            if (!myWaterSplash.emit && myTransform.position.y <
yWater) {
                myWaterSplash.emit = true;
            }

            // If we are totally into water, reset variables,
and transport penguin to be closer to the destiny
            if (timer >= 1) {
                divingDir = new Vector3 (divingDir.x,
JUMPING_SPEED, divingDir.z);
                divingState = DivingState.GoingOutWater;
                timer = 0;
```

```
                myTransform.position = new Vector3
(placeToGoNowDiving.x -
divingDir.normalized.x*RECOVERY_DISTANCE,
myTransform.position.y, placeToGoNowDiving.z -
divingDir.normalized.z*RECOVERY_DISTANCE);
                origin = myTransform.position;
            }
            break;

        case DivingState.GoingOutWater:

            // Jump out of the water to the destiny level
            myTransform.position = Vector3.Lerp(origin, new
Vector3(placeToGoNowDiving.x, yPos, placeToGoNowDiving.z),
timer);

            // Splash particles
            if (myWaterSplash.emit && myTransform.position.y >
yWater - 1.1f) {
                myWaterSplash.emit = false;

myWaterSplash.GetComponent<Transform>().localPosition -=
transform.up;
            }

            // Stop animation when needed
            if (timer >= 0.4f)
                myAnimator.SetBool ("Jumping", false);

            // If we reached the desired position
            if (myTransform.position == new
Vector3(placeToGoNowDiving.x, yPos, placeToGoNowDiving.z)) {

                // Reset values
                divingState = DivingState.Nothing;
                timer = 0;
                currentLevel = nextLevelToGoDiving;
                myPreviousLevel = currentLevel.GetPreviousLevel
();
                myNextLevel = currentLevel.GetNextLevel ();
                myTransform.eulerAngles = new Vector3 (0, 180,
0);
                divingValuesSet = false;

                // If penguin had enqueued another place to go
diving once this one is finished, start process again
                if (nextLevelToGoAfterDiving){
                    SetDestiny (nextLevelToGoAfterDiving,
false);
                    nextLevelToGoAfterDiving = null;
                }
            }
            break;
        }
    }
}
```

Code is long and complex, but easy to understand by reading the comments. It works awesome, the animated penguin cheers up the whole scene.

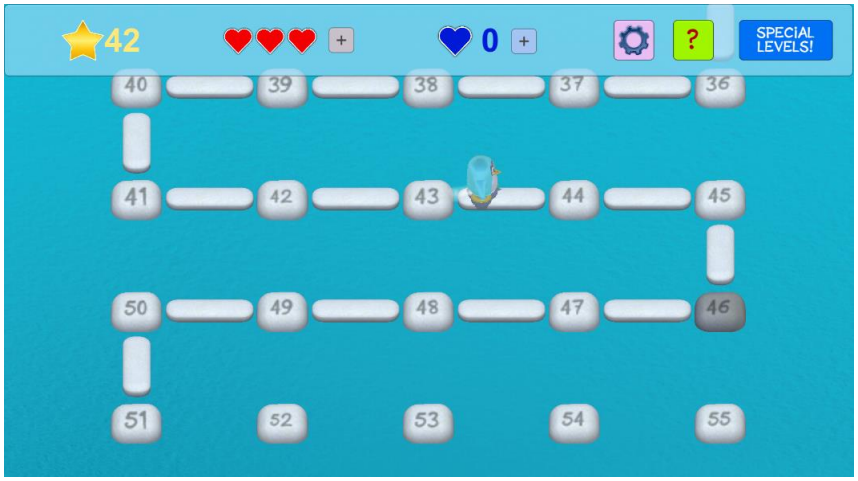

Fig. 27 - Penguin walking from level 43 to the selected level 46

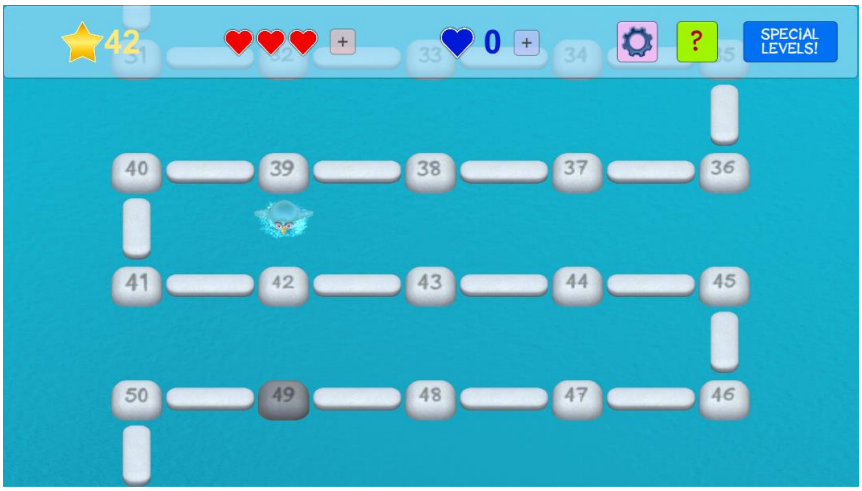

Fig. 28 - Penguin starting to dive from level 39 to the selected level 49

4.11 Monetization

Monetization strategy is described in the corresponding chapter of the Game Design Document. Here, we will see how to implement both money earning systems: IAP (In-app Purchases) [8] and Rewarded Video Ads [7].

4.11.1 IAP (In-app Purchases)

By using IAP, "the Google Play app conveys billing requests and responses between your application and the Google Play server. In practice, your application never directly communicates with the Google Play server. Instead, your application sends billing requests to the Google Play app over interprocess communication (IPC) and receives responses from the Google Play app. Your application does not manage any network connections between itself and the Google Play server" [8].

Fig. 29 - Google Play IAP pop up

Unity3D has its own service and plugin to integrate IAP into games [28], and automatically treat it as Google Play IAP if the game is built for Android. The process to integrate it is the following:

First, enabling IAP service into the Unity3D project. This will also automatically enable Analytics service, which may sound totally friendly, but its background objective is to let Unity know your revenue, and then force you to buy Unity Premium if your income is higher than the minimum permitted with Unity Free. Once this is completed, we are able to import the IAP plugin.

Now, let's see how we have coded our purchaser script:

```
/ The Unity Purchasing system.
private static IStoreController m_StoreController;
// The store-specific Purchasing subsystems.
private static IExtensionProvider m_StoreExtensionProvider;

// Our products to be purchased
public static string kProductIDConsumableTenBlueHearts = "blue";
public static string kProductIDNonConsumableGoldenHeart =
"golden";

void Start()
```

```
{
    // Default purchaser initializations
    // ...
}

public void InitializePurchasing()
{
    // Default purchaser initializations
    // ...

    // We add our products to the builder, and set them as
consumable or non consumable
    builder.AddProduct(kProductIDConsumableTenBlueHearts,
ProductType.Consumable);
    builder.AddProduct(kProductIDNonConsumableGoldenHeart,
ProductType.NonConsumable);

    // Default purchaser initializations
    // ...
}


private bool IsInitialized()
{
    //...
}

// Buying functions to be called from outside
public void BuyTenBlueHearts(){
    BuyProductID(kProductIDConsumableTenBlueHearts);
}
public void BuyGoldenHeart(){
    BuyProductID(kProductIDNonConsumableGoldenHeart);
}

void BuyProductID(string productId)
{
    // Default purchaser function
}

public void OnInitialized(IStoreController controller,
IExtensionProvider extensions)
{
    // Default purchaser function
}


public void OnInitializeFailed(InitializationFailureReason
error)
{
    // Show error
    // ...
}

public PurchaseProcessingResult
ProcessPurchase(PurchaseEventArgs args)
{
    // Management after player successfully bought ten blue
hearts
    if (String.Equals (args.purchasedProduct.definition.id,
kProductIDConsumableTenBlueHearts, StringComparison.Ordinal)) {
```

```
((BlueShop)GameObject.FindObjectOfType(typeof(BlueShop))).BuyTen
BlueHeartsFromPurchaser();
    }

    // Management after player successfully bought the golden
heart
    else if (String.Equals (args.purchasedProduct.definition.id,
kProductIDNonConsumableGoldenHeart, StringComparison.Ordinal)) {

((RedShop)GameObject.FindObjectOfType(typeof(RedShop))).BuyGolde
nHeartFromPurchaser();
    }

    // Unknown product ID, show error
    else {
        //...
    }

    return PurchaseProcessingResult.Complete;
}


public void OnPurchaseFailed(Product product,
PurchaseFailureReason failureReason)
{
    // Show error
    // ...
}
```

Now the last step would be to select the corresponding price to each product ID into the Google Play Developer Consoler, and that price will automatically pop up when a “Buy” function of this purchaser is called.

4.11.2 Rewarded Video Ads

Rewarded Video Ads are "video ad units allow developer to reward users with in-app items for watching video ads. They may be served only after a user explicitly chooses to view a rewarded ad. This puts the user in control of their in-app experience. Developer can specify the reward values associated with the ad units in his or her app and set different rewards for different ad units. Users will receive the reward for viewing the video ad without needing to install anything" [7].

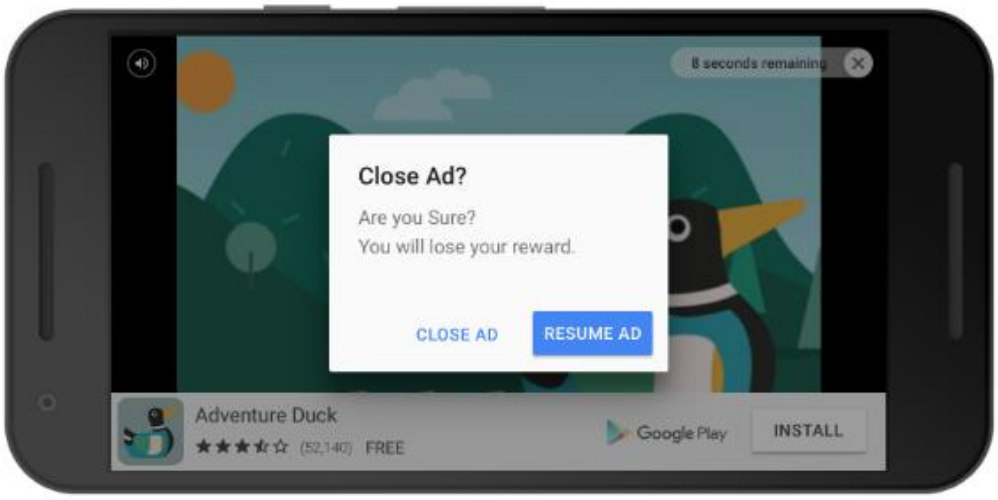

Fig. 30 - Rewarded Videa Ad, player could close it but losing the reward

As happened with IAP, there is also a plugin to integrate Rewarded Video Ads into Unity3D by using Admob [29]. There are also many intermediates that offer more services and ads types variety, but we only want Rewarded Video Ads so Admob has been considered the best option.

One important thing to take into account is that a video ad must first be loaded, and then played. We have here an important decision: Do we load the video once player selects to watch it, or do we load a video at the beginning of the game to have it prepared to be instantly watched when selected? Obviously, the second option is better for gameplay experience, as player will never have to wait. However, we must keep in mind one extra limitation: online data usage. When a video is loaded, it can take 10 or 20 megabytes of download, which may be too much for people who only have 100 or 500 megabytes available a month. These people will try to watch videos only with wi-fi connection, so we must not force the loading of the videos if they do not choose to watch them. We have considered this case as priority, so ad videos will be loaded only once players touches the button to watch them, and after the having be loaded, they will be automatically played.

Next is our implementation:

```
// Default global variables
//...

void Start(){

    // Default initializations
    //...

    // We could request to load the video from the very
beginning, but we will not do so for data usage
    //RequestRewardBasedVideo ();
}

// This is the function that we will call form outside
public void ShowRewardedVideo(){

    // We show the Loading Screen
    SceneManager.LoadScene ("Loading...",
LoadSceneMode.Additive);

    // And then we start loading the video
    RequestRewardBasedVideo ();
}

private void RequestRewardBasedVideo()
{
    // Example test Id for android rewarded video ads
    string adUnitId = "ca-app-pub-3940256099942544/5224354917";

    // Load the video
    AdRequest request = new AdRequest.Builder().Build();
    rewardBasedVideo.LoadAd(request, adUnitId);
}

// Once the video is loaded, automatically show it
public void HandleRewardBasedVideoLoaded(object sender,
EventArgs args)
{
    rewardBasedVideo.Show ();
}

// Once the video has been opened or failed to load, simply
unload the Loading Screen
public void HandleRewardBasedVideoFailedToLoad(object sender,
AdFailedToLoadEventArgs args)
{
    SceneManager.UnloadSceneAsync ("Loading...");
}
public void HandleRewardBasedVideoOpened(object sender,
EventArgs args)
{
    SceneManager.UnloadSceneAsync ("Loading...");
}

// If player has watched the full video, give the corresponding
reward
public void HandleRewardBasedVideoRewarded(object sender, Reward
args)
{
```

```
((RedShop)GameObject.FindObjectOfType(typeof(RedShop))).ObtainTw
oRedHeartsFromRewardedVideoAd ();
}
```

With this developed script, from now on it is only needed to call the function "ShowRewardedVideo()" and everything will be managed: video will be loaded with a loading screen, next it will be shown, and player will receive the reward just by watching the full video.

4.12 Title Screen

We wanted a background picture for the title screen of the game that denotes dynamism and that easily shows how the game looks like and how it will played. We do so starting from a screenshot of the game and improving it with Photoshop to obtain the desired result.

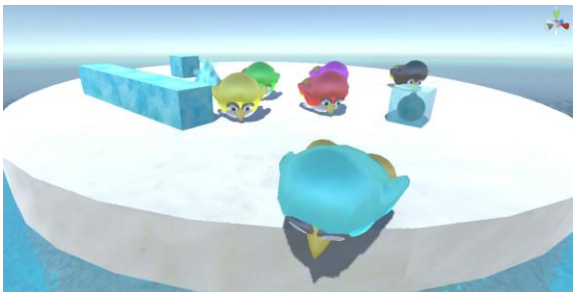

Fig. 31 - Title Screen: screenshot we part from

To obtain a dynamic sensation, we distort the picture:

Fig. 32 - Title Screen: distorted

To fit usual screens, we make it higher and draw the new borders:

Fig. 33 - Title Screen: higher

Next, we improve every single aspect of the image and defects caused by distortion, such as the sea at the top left corner or the hardness or the color of the penguins:

Fig. 34 - Title Screen: details improved

Next, we use a color layer to give penguins more vivid colors.

Fig. 35 - Title Screen: vivid colors

Now we must fix a huge error, the aliasing and the hardness of some borders. We do it manually with the defocusing tool, and then we focus the picture a bit to see it better.

Fig. 36 - Title Screen: Antialiasing

Now we are ready to place the title of the game. We use some text parameters to make it more beautiful.

Fig. 37 - Title Screen: title text

Finally, we add a special effect offered by Photoshop: a flash, behind the text of the title:

Fig. 38 - Title Screen: flash special effect

Now that we have the title screen background, we must give it functionality. We add a "start" button, that will normally lead you to the map of levels, but it will do other thing before in two specials occasions:

- If it is the first time player is running the game, he or her will be offered to enter the nick of the person who may had recommended the game to him or her, and by doing so player will earn 2 blue hearts.

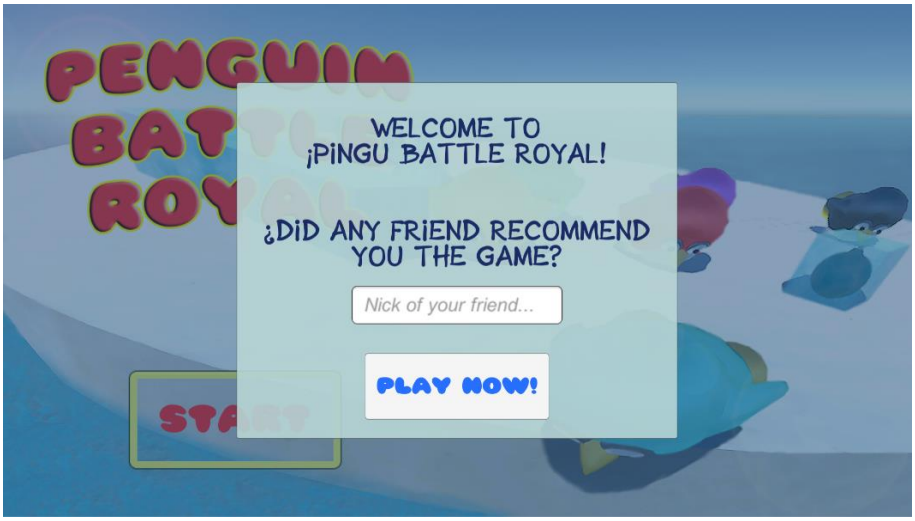

Fig. 39 - Title Screen: first time playing

- If a friend who you recommended the game did entered your nick at him or her first time playing, and he or her already reached level 10, you will be told so and you will earn 8 blue hearts.

Fig. 40 - Title Screen: Reward because of friend playing

Implementation of that online communication will be treated in the next chapter.

4.13 Sharing implementation

Even when the game is not multiplayer, it is desired it to have a sharing community to easily get new players. Players must be able to share it with their friends, and they will be tempted to do so because when a friend reaches level 10, player will earn 8 blue hearts. How can we do so? We need to keep an online storage and access to it to know when players must be rewarded and when not.

4.13.1 Online Storage

Players are designed by nicks. Nicks must not be repeated, so it is necessary an online storage to know which nicks are already in use.

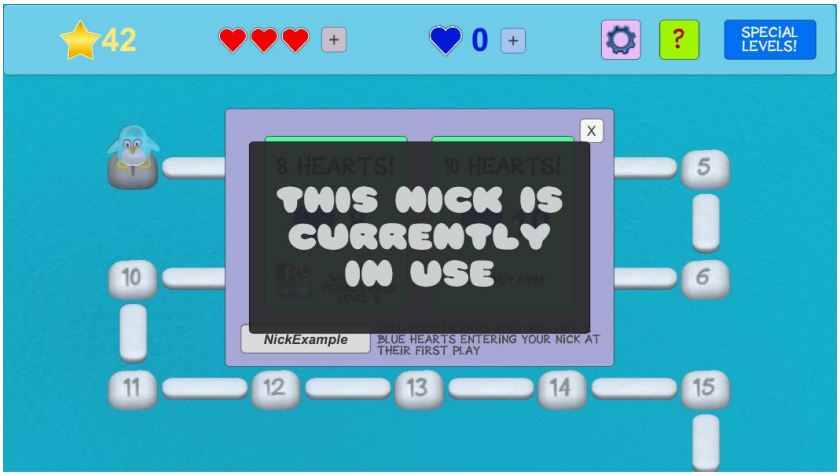

Fig. 41 - Checking entered nick is already in use

We also need to have a global source to tell it “hey, I reached level 10, so the player <nick> who recommended the game to me needs to earn 8 blue hearts”, and ask it “hey, did any of my friends who I recommended the game already reach level 10?”.

In less words, we need to save this information online:

- Every nick that is already in use
- Every nick that will receive a rewards next time playing

We can establish it as two lists of strings, serialize them, and then uploading that serialized information to an online text file. Usually, videogames use online databases, but we want to save just a few of simple information, so it is no needed to use a complex database.

The way to store information online into a text file is to have a .txt file into a www host, and a .php script that may modify that .txt online when desired. We will have a simple .php that receives a string and writes that string into the .txt of the online host:

```
<?php
$txtcontent = $ REQUEST['txt'];
$fp = fopen('MySeverSata.txt', w);
fwrite($fp, $txtcontent);
fclose($fp);
?>
```

Now, let's see how do we call that .php from Unity3D:

```
private IEnumerator WriteTextToWWW (string text){

    // We encrypt the content to avoid server hacking
    text = Encrypt(text);

    // Call the .php and give it our text as parameter
    WWW www = new WWW(urlPHP + "?txt=" + text);

    // Wait for finishing the process
    yield return www;

    if (www.error != null)
    {
        // Manage error
    }
    else
    {
        // Manage success
    }
}
```

For reading the text file we don't even need a .php. It is a direct request:

```
private IEnumerator GetTextFromWWW ()
{
    // Download the content of text file
    WWW www = new WWW(urlTextFile);

    // Wait for finishing the process
    yield return www;

    if (www.error != null)
    {
        // Manage error
    }
    else
    {
        // Get the text from the WWW
        textFromWWW = www.text;

        // Decrypt the text
        textFromWWW = SaveLoadData.Decrypt(textFromWWW);

        // Manage success
    }
}
```

Our server manager is ready. We only need now to assign the proper calls to these functions to obtain the results mentioned before. Easy work.

4.13.2 Share on Social Networks

What we want is that functionality that some apps have which consists on a pop up that shows every single via that player's android device has for sharing information (mostly social networks), and being able to set a predefined text message to share once one of those via has been selected.

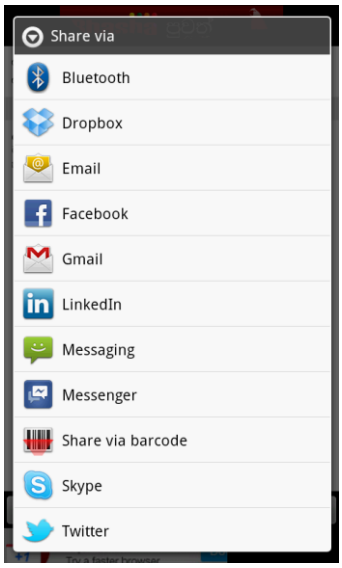

Fig. 42 - Android share function on selected via

The way we have implemented that sharing function is the following:

```
public void ShareButton(){

    // We directly get the text to share from the Csv
Localization file
    string mytext =
AutoLocalizingText.GetString("Share.Message");

    // Insert player nick and download link into the message to
share
    mytext = mytext.Replace ("<nick>", "\"" + SavedData.myNick +
"\"");
    mytext = mytext.Replace ("<link>", downloadLink);

    Share (mytext);
}


private void Share(string textToShare){

    // Initialize an intent android java class
    AndroidJavaClass intentClass = new AndroidJavaClass
("android.content.Intent");
```

```
    // Initialize a intent android java object and insert into
it the information of the activity of sending a text
    AndroidJavaObject intentObject = new AndroidJavaObject
("android.content.Intent");
    intentObject.Call<AndroidJavaObject> ("setAction",
intentClass.GetStatic<string> ("ACTION_SEND"));
    intentObject.Call<AndroidJavaObject> ("setType",
"text/plain");
    intentObject.Call<AndroidJavaObject> ("putExtra",
intentClass.GetStatic<string> ("EXTRA_SUBJECT"), "SUBJECT");
    intentObject.Call<AndroidJavaObject> ("putExtra",
intentClass.GetStatic<string> ("EXTRA_TEXT"), textToShare);

    // Initialize a android java class based on Unity
    AndroidJavaClass unity = new AndroidJavaClass
("com.unity3d.player.UnityPlayer");

    // Initialize a new android java object to work as the
current activity of Unity
    AndroidJavaObject currentActivity =
unity.GetStatic<AndroidJavaObject> ("currentActivity");

    // Use it to call the activity of sending a text initialized
before
    currentActivity.Call ("startActivity", intentObject);
}
```

The text that we want to share, stored into the Csv Localization file, is: “Join Penguin Battle Royal! Play it now on Android and enter my nick <nick> to start with 2 blue hearts! <link>”. It is possible to share predefined text with most of the social networks, but it is actually don’t allowed via Facebook, due to its policy program. We hope players sharing on facebook will write a proper text to invite friends to play it. Nevertheless, although facebook will not show the text, it will actually show the download link of the game. That is more than nothing. Everything seems working fine with the rest of social networks that has been tested.

4.14 Performance

This game is intended to be player in many android devices, even those not very powerful. Therefore, we need to care about performance. We do it in different ways:

Memory: It is desired a apk with a low megabyte size, as some android devices does not have enough free space. We also want that RAM memory not being too overloaded. Another important thing is that low sized scenes will be loaded faster than heavy ones. In order to all three objectives, let's reduce the size of the heaviest component of most games: the textures.

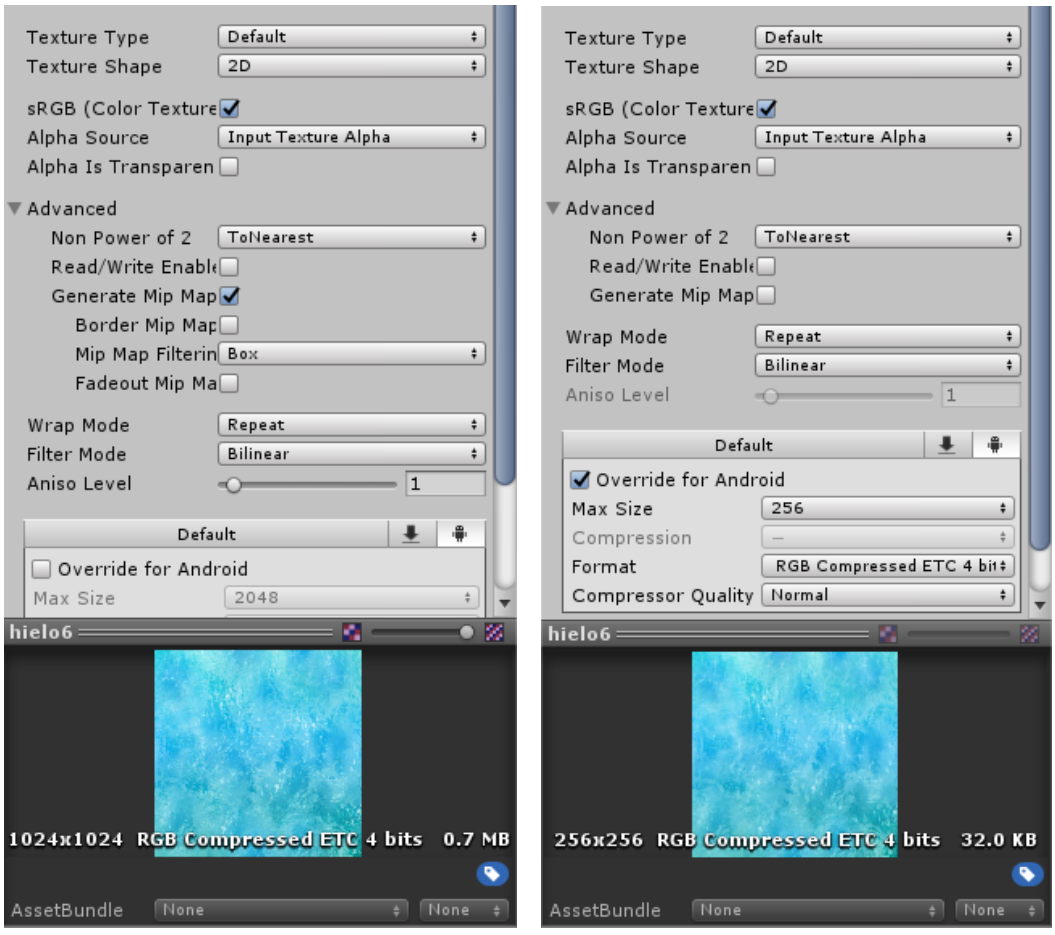

Fig. 43 - Same texture before and after compression: From 700KB to 32KB

There are two easy ways to reduce a texture size into Unity3D:

- Use the default compression tool of Unity: It is very efficient, but we must check which is the minimum size at which our texture still looks right. At our example in our figure above, it was 256x256.

- Do not generate Mip Maps: Mip Maps are different size versions of the same textures designed to be used for different LODs (Level of Detail). It is very useful in

games with scenes that have many elements far away, but our game is not the case. We save valuable memory if we do not generate Mip Maps.

Another important thing to take into account for performance are the Draw Calls [4]. In simple words, a Draw Call is the process to read and then draw a single texture or sprite intro screen, and the number of Draw Call at a specific moment is the number of draws that is doing the GPU, one per each texture or sprite into screen. Draw Calls are very expensive for performance, so we need no low them as much as possible. Unity3D already has an automatic performance function called Batching [4], which tries group Draw Calls of the same texture or sprite to be as expensive as if it where only one, although it only works totally right with static game objects.

We are going to use 2 strategies to low the number of Draw Calls:

- Deactivate shadows by default: Shadows duplicate the number of Draw Calls. They are amazingly expensive. We have deactivated them by default, and we let players activate them in the options menu if they consider their android device is powerful enough.

- Using an Atlas [30]: An Atlas is a single texture that contains many textures distributed inside it. Unity can read a single Atlas and use its sub-textures as independent textures to be used into the game. Even when drawing different textures, if all of them come from the same Atlas, the GPU cost if one single Draw Call, as it is reading only one texture: our Atlas. We could perfectly create our own Atlas with Photoshop by packing all of our textures into a single transparent square, but we are going to choose other way: The Unity Sprite Packer [31]. It is a powerful tool that automatically packers the chosen sprites imported into a built-in Atlas.

The way to select which sprites must Sprite Pacer pack when the into the same Atlas is to write a same Packing Tag into the settings of all of them:

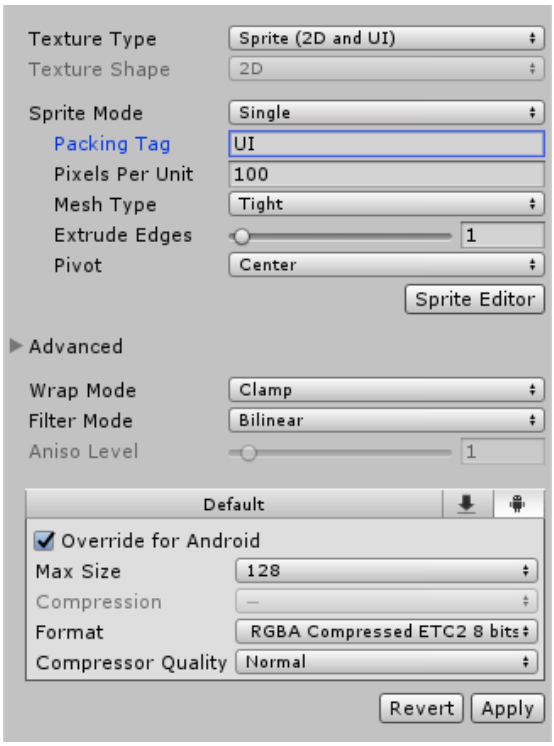

Fig. 44 - Writting a Packing Tag to be packed into the "UI" Atlas of the Sprite Packer

Now that we have the same Packing Tag in all our desired sprites, let's to the Sprite Packer window and press the Pack function:

Fig. 45 - Sprite Packer: our UI Atlas

As a last objective, we want to reduce scene loading tines. Each time we load a new scene, every object of the current scene is deleted from the RAM memory, and every object of the new scene is then loaded into the RAM memory. This happens even when the new scene has elements that the scene before also has. Unity doesn't care, it deletes them and load them again.

What we want is to keep every object we are going to need into memory for not being deleted and reloaded when changing scene (which is very frequent since each level is a different scene). We will do so with a singleton [32], a script that will have one single instance and that will not be destroyed on scene loading, and we want that script keeping references to every element that we do not want to be unloaded. Unity loads in memory not just game object in scene but also those objects that are referenced from any script component in scene, so we can achieve our objective with just references to the desired objects. Next is our singleton implementation:

```
//  Instance reference
private static SingletonHolder _instance;

void Awake()
{
    if(_instance == null)
    {
        // I am the instance of this singleton
        _instance = this;

        // I will not be destroyed
        DontDestroyOnLoad(this.gameObject);
    }
    else
    {
        // I am not the instance of this singleton, I must be
destroyed
        Destroy(this.gameObject);
    }
}

// References to every object that we do not want to be unloaded
when loading a new scene
public Object[] ObjectsHolder;
```

We only need now to add this script component to an empty game object at the beginning of the game, and then drag to the public array “ObjectsHolder” every object that we do not want to be unloaded when loading a new scene. Results are better than expected: loading times that took about five seconds take now less than one second. Objective achieved.

5. Project Monitoring

The project has been developed under supervision of tutor Miguel Chover Selles. Fortnightly reports were sent to him to check the project progress, receiving advices and guidelines of how to schedule the invested time.

Every listed task on chapter "2. Planning" has been performed, most of them surrounding the predicted time. One new task devised to improve the project has been added during the development period: implementation of online storage, which will be explained at next chapter. It took about 12 hours to be finished. Total invested time is about 300 hours.

There have been few differences from the original planning:

- The writing of this final report did not start once the project was finished as initially planned, but at the very beginning of its development, as adviced by tutor. It was especially useful to write de Game Design Document to make objectives clearer.
- The time invested into writing this final report has exceeded the planned 25 hours by far. Because of this, and also because of the new added task, the planned task "Design and implement a high number of levels" has been developed in less than 22 hours as planned. There have been developed just 11 levels for the presentation of this project in July, which illustrate perfectly every aspect of the game. More of them will be developed as future work.
- The project has been developed while a stay in practices of unexpected full-time (8 hours a day of practices, plus 1 hours of break to eat, plus half an hour twice a day of displacement from home to the business). Because of this reason, project was not able to be developed with a regular schedule as planned, but investing the available hours every night and completely full time on weekends. It has been a hard work, but satisfactorily overcome.

6. Conclusions

6.1 Objectives and Goals Achieved

Proudly, every single objective raised at the beginning of this technical report has been achieved:

- It has been developed a videogame for android devices that can be controlled via either gyroscope or accelerometer, resulting in an almost identical game experience.
- Nice looking characters, items and levels have been modeled, animated and implemented:

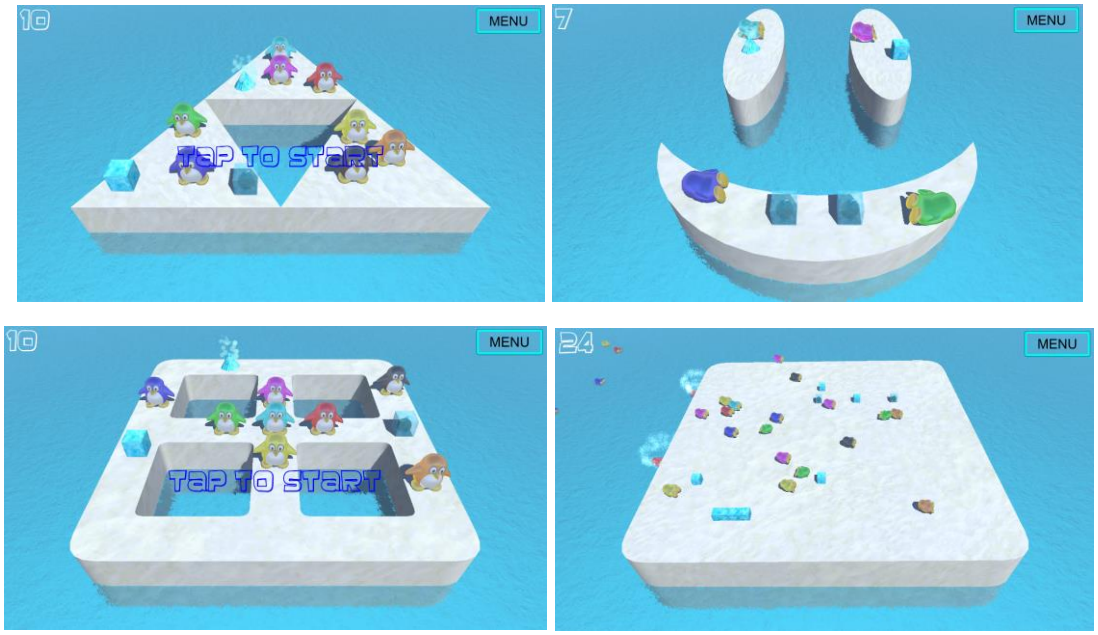

Fig. 46 - Levels Screenshots

- Android builds audio delay with Unity has been fixed.
- Save data has been safely stored in a file with encrypted.
- Csv localization has been developed and integrated.

	key	English	Spanish
1			
2	Word.Music	Music	Música
3	Word.Waves	Waves	Oleaje
4	Word.Mute	Mute	Silenciado
5	Word.Effects	Effects	Efectos
6	Word.Unmuted	Unmuted	No Silenciado
7	Word.Enabled	Enabled	Activas
8	Word.Disabled	Disabled	Inactivas
9			
10	Word.Yes	Yes	Sí
11	Word.No	No	No
12			
13	Game.StartButton	Tap to start	Toca para empezar
14	Game.Failed	You failed...	Has perdido...
15	Game.Win	You win!	¡Has ganado!
16	Game.Time	Time:	Tiempo:
17	Game.RECORD	NEW RECORD!	¡NUEVO RECORD!
18	Game.AccelAlert	Alert! Excessive in	Alerta! Inclinación exc
19			
20	Word.Loading	Loading	Cargando
21			

64	Warning.SpecialLe	Pass level 10 to un	Cuando superes el niv
65	Warning.Exit	Exit game?	¿Quieres salir del jueg
66	Warning.Internet	You need internet c	Necesitas conexion a
67	Warning.Server	Server error. Try ag	Error de conexion con
68	Warning.NickUsed	This nick is current	Este nick esta actualm
69	Warning.EnterNick	Enter a nick below	Escribe un nick debajo
70	Warning.CheckNicl	"nickreplace"\n¿Is i	"nickreplace"\n¿Es co
71	Warning.Sure	Sure!	¡Claro!
72	Warnink.Recheck	Wait, I'll check it ag	Espera que lo revise..
73			
74	Warning.NickUnuse	There isn't any play	No hay ningun jugado
75	Main.Welcome	Welcome to\n¡Ping	Bienvenido/a a\n¡Ping
76	Main.EnterFriendN	Nick of your friend.	Nick de tu amigo...
77	Main.PlayNow	Play now!	¡Jugar ya!
78	Main.Congratulatio	Congratulations! A	¡Felicidades! Un amig
79	Main.Great	Great!	¡Genial!
80			
81	Share.Message	Join Penguin Battle	¡Únete a Penguin Batt

Fig. 47 - Localization Csv file with English and Spanish for now

- The map of levels is dynamic and interactive enough to be so charming as levels may be.

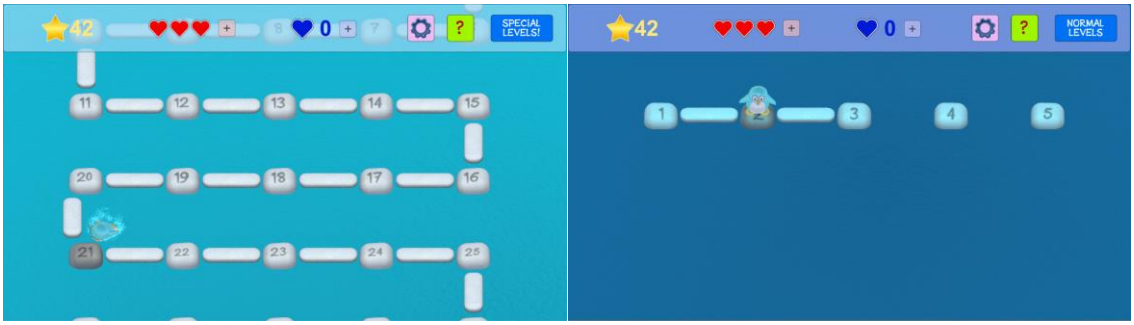

Fig. 48 - Map of levels and special levels

- The game has been nicely monetized to be able to be played for free but tempting player to buy hearts.

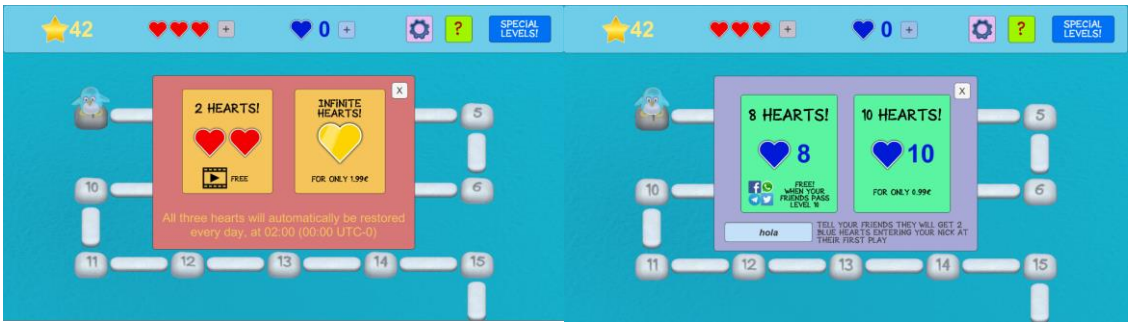

Fig. 49 - Shops of red hearts and blue hearts

- It has been implemented the function of "sharing on social networks"
- The game works fluid, with no bugs at all, and with an improved performance for working rightly with any android device.

Fig. 50 - Main Screen of Penguin Battle Royal

6.2 Added Aspects

When dealing with sharing on social networks, it was noticed that we should reward players for sharing. However, the simple action of sharing should not be enough to receive a reward, because some players could share it in a closed whatsapp group (for example) receiving then a reward for nothing. It was decided that rewards should only be received when friends join the game because of a recommendation, and they reach a semi-advanced level as could be level ten. This seems easy at the beginning but being realistic this is only possible with an online storage where players and recommendations and registered.

Implementation of an online storage system looked necessary, so it was added to the tasks and correspondingly developed. It works perfectly with the small group of connected devices that was tried, so let's see how it may work with thousands of players. As explained in next sub-chapter, an improvement of this system must be done as future work, but current the implementation was worthy without doubts and has improved the qualities of the project.

6.3 Future and Project possibilities

This project is intended to become a real full game with the required quality, so there is some work left that must be done as next future work:

- First thing of all, many more levels must be developed. For being a launchable game, it should have at least one hundred levels. They must be designed with a smart progression of player learning and inclusion of new elements. Even with a launched game, new levels and in-game elements should be developed as updates to keep players playing.
- For now, the game is localized to English and Spanish. Localization implementation is ready to insert new languages, so it is future work to fill the Csv file with as many languages as possible to reach the widest public.
- Accelerometer gameplay experience is identical gyroscope's when the device initial position is parallel to the floor. It is future work to try making it work identical 360 degrees. It seems hard to achieve must probably not impossible.
- The online storage format is currently a txt file. For future work, it may be used a database instead, as having thousands of players it results more efficient and secure to make SQL requests than reading and writing onto a txt file.

7. Bibliography

[1] "Unity3D," [Online]. Available: http://www.unity.com. [Accessed Feb 2017].

[2] "Angry Birds," [Online]. Available: https://www.angrybirds.com. [Accessed Feb 2017].

[3] "Audio Source Playing Clip After Delay On Build," [Online]. Available: https://www.reddit.com/r/Unity3D/comments/40n1ru/audio_source_playing_clip_after_delay_on_build. [Accessed Feb 2017].

[4] Unity3D, "Draw Call Batching," [Online]. Available: https://docs.unity3d.com/Manual/DrawCallBatching.html. [Accessed May 2017].

[5] Google Play, "Sensors Motion," [Online]. Available: https://developer.android.com/guide/topics/sensors/sensors_motion.html. [Accessed Jun 2017].

[6] "Audio Source Playing Clip After Delay On Build," [Online]. Available: https://www.reddit.com/r/Unity3D/comments/40n1ru/audio_source_playing_clip_after_delay_on_build. [Accessed Feb 2017].

[7] AdMob, "Overview of rewarded video ad units," [Online]. Available: https://support.google.com/admob/answer/7372450. [Accessed May 2017].

[8] Google Play, "In-app Billing Overview," [Online]. Available: https://developer.android.com/google/play/billing/billing_overview.html. [Accessed May 2017].

[9] R. Goodrich, "LiveScience," [Online]. Available: https://www.livescience.com/40103-accelerometer-vs-gyroscope.html. [Accessed Jun 2017].

[10] Unity3D, "Gyroscope Documentation," [Online]. Available: https://docs.unity3d.com/ScriptReference/Gyroscope.html. [Accessed Jun 2017].

[11] Unity, "Vector3 Documentation," [Online]. Available: https://docs.unity3d.com/ScriptReference/Vector3.html. [Accessed Jun 2017].

[12] Unity3D, "Quaternion Documentation," [Online]. Available: https://docs.unity3d.com/ScriptReference/Quaternion.html. [Accessed Jun 2017].

[13] Unity, "Acceleration Documentation," [Online]. Available: https://docs.unity3d.com/ScriptReference/Input-acceleration.html. [Accessed Jun 2017].

[14] Unity Wiki, "Low Pass Filter," [Online]. Available: http://wiki.unity3d.com/index.php/Low_Pass_Filter. [Accessed Jun 2017].

[15] Unity, "Lerp Documentation," [Online]. Available: https://docs.unity3d.com/ScriptReference/Vector3.Lerp.html. [Accessed Jun 2017].

[16] Unity, "Water in Unity Documentation," [Online]. Available: https://docs.unity3d.com/Manual/HOWTO-Water.html. [Accessed Jun 2017].

[17] Unity, "Raycasting Documentation," [Online]. Available: https://docs.unity3d.com/ScriptReference/Physics.Raycast.html. [Accessed Jun 2017].

[18] Unity, "Particle System Documentation," [Online]. Available: https://docs.unity3d.com/ScriptReference/ParticleSystem.html. [Accessed Jun 2017].

[19] Unity, "Ellipsoid Particle Emitter Documentation," [Online]. Available: https://docs.unity3d.com/Manual/class-EllipsoidParticleEmitter.html. [Accessed Jun 2017].

[20] Android Developers Documentation, "Android Sound Pool," [Online]. Available: https://developer.android.com/reference/android/media/SoundPool.html. [Accessed Jun 2017].

[21] Fadden, "Android Audio Bypass," [Online]. Available: https://github.com/fadden/android-audio-bypass. [Accessed Jun 2017].

[22] Apache, "Apache 2.0 License," [Online]. Available: https://www.apache.org/licenses/LICENSE-2.0. [Accessed Jun 2017].

[23] Unity3D, "Streaming Assets Documentation," [Online]. Available: https://docs.unity3d.com/Manual/StreamingAssets.html. [Accessed Jun 2017].

[24] Unity3D, "Persistent Data Path Documentation," [Online]. Available: https://docs.unity3d.com/ScriptReference/Application-persistentDataPath.html. [Accessed Jun 2017].

[25] Unity3D, "Player Prefs Documentation," [Online]. Available: https://docs.unity3d.com/ScriptReference/PlayerPrefs.html.

[26] Unity3d, "Json Documentation," [Online]. Available: https://docs.unity3d.com/Manual/JSONSerialization.html. [Accessed Jun 2017].

[27] Unity3D, "System Language Documentation," [Online]. Available: https://docs.unity3d.com/ScriptReference/SystemLanguage.html. [Accessed Jun 2017].

[28] Unity3D, "Intregating Unity IAP," [Online]. Available: https://unity3d.com/es/learn/tutorials/topics/ads-analytics/integrating-unity-iap-your-game. [Accessed Jun 2017].

[29] Admob, "Rewarded Video Ads at Unity3D," [Online]. Available: https://developers.google.com/admob/unity/rewarded-video. [Accessed Jun 2017].

[30] Wikipedia, "Texture Atlas," [Online]. Available: https://en.wikipedia.org/wiki/Texture_atlas. [Accessed Jun 2017].

[31] Unity3D, "Sprite Packer Documentation," [Online]. Available: https://docs.unity3d.com/Manual/SpritePacker.html. [Accessed Jun 2017].

[32] Wikipedia, "Singleton pattern," [Online]. Available: https://en.wikipedia.org/wiki/Singleton_pattern. [Accessed Jun 2017].